Titre: Title:	Caractérisation de projets de développement logiciel dans une perspective de flux de connaissances
Auteur: Author:	Olivier Gendreau
Date:	2010
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Gendreau, O. (2010). Caractérisation de projets de développement logiciel dans une perspective de flux de connaissances [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/391/

UNIVERSITÉ DE MONTRÉAL

CARACTÉRISATION DE PROJETS DE DÉVELOPPEMENT LOGICIEL DANS UNE PERSPECTIVE DE FLUX DE CONNAISSANCES

OLIVIER GENDREAU

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION

DU DIPLÔME DE PHILOSOPHIAE DOCTOR (Ph.D.)

(GÉNIE INFORMATIQUE)

AOÛT 2010

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

CARACTÉRISATION DE PROJETS DE DÉVELOPPEMENT LOGICIEL DANS UNE PERSPECTIVE DE FLUX DE CONNAISSANCES

présentée par : GENDREAU Olivier

en vue de l'obtention du diplôme de : Philosophiae Doctor

a été dûment acceptée par le jury d'examen constitué de :

M. GAGNON Michel, Ph.D., président

M. ROBILLARD Pierre N., Ph.D., membre et directeur de recherche

M. DESMARAIS Michel, Ph.D., membre

M. LETHBRIDGE Timothy, Ph.D., membre

REMERCIEMENTS

Mes premiers remerciements vont à mon directeur de recherche, Pierre N. Robillard. Son indéfectible soutien au cours des cinq dernières années a été crucial au succès de cette aventure doctorale. Les innombrables commentaires et suggestions qu'il a pris le soin de formuler m'ont poussé à produire un travail dont je suis aujourd'hui fier. Je tiens particulièrement à souligner ses grandes habiletés de motivateur, qui m'ont incité, contre vents et marées, à mener à bien mes études doctorales.

Je désire aussi remercier tous les participants des projets intégrateurs en génie logiciel à l'École Polytechnique entre la session d'hiver 2006 et d'hiver 2009 ayant accepté que j'utilise leurs jetons d'activités, qui sont à la base de la méthodologie de cette thèse.

Merci à mon collègue des deux premières années, Simon. Son calme, sa rigueur et son désir d'excellence m'ont inspiré dans mon parcours.

Merci à mes amis de m'avoir soutenu et un merci particulier à Valérie d'avoir pris le temps de corriger mes articles en anglais, et ce, malgré son horaire extrêmement chargé.

Merci à mes parents, René et Johanne, ainsi qu'à mon frère Pierre-Samuel de m'avoir toujours encouragé dans mes projets et en particulier lors des difficiles cinq dernières années.

Un merci tout spécial à ma complice de tous les jours, An. Merci d'être mon rayon de soleil, qui a su éclairer mes journées plus sombres. Ta compréhension et ton amour m'ont porté jusqu'à la fin de mon doctorat.

RÉSUMÉ

Face aux ratés auxquelles fait face l'industrie du développement logiciel et l'incapacité des différentes approches de processus logiciels à régler ces problèmes, il s'avère intéressant de se baser, d'une part, sur les acquis du domaine des processus logiciels et, d'autre part, de s'inspirer des innovations de domaines connexes. En particulier, la gestion des connaissances appliquée au génie logiciel est un sujet présentement en émergence. Une meilleure compréhension des mécanismes de création et de conversion de connaissances au sein d'un projet de développement logiciel est une avenue de recherche prometteuse. L'objectif principal de cette thèse de doctorat est donc de caractériser les projets de développement logiciel dans une perspective de flux de connaissances.

Cette thèse par articles propose d'atteindre l'objectif de recherche par la présentation de trois articles en plus d'un chapitre détaillant des résultats complémentaires.

Le premier article présente et justifie la méthodologie utilisée dans le cadre des travaux de recherche de cette thèse. Plus précisément, l'article détaille la méthodologie ATS (*activity time slip*), une approche à partir de laquelle des développeurs logiciels doivent enregistrer leurs activités dans une perspective de connaissances. Les données recueillies sont ensuite codifiées selon le modèle de flux de connaissances, qui est inspiré du modèle de création de connaissances de Nonaka & Takeuchi (1995) et qui définit six facteurs cognitifs dans le cadre d'un projet de développement logiciel: l'acquisition, la cristallisation, la validation, la réalisation, la vérification et l'organisation du travail. Une étude de cas multiples est présentée, afin de démontrer l'originalité et la pertinence de la méthodologie proposée.

Le second article présente l'utilisation du modèle de flux de connaissances, dans le cadre d'une étude de cas, afin d'analyser les conséquences de la qualité de la documentation lors de la réutilisation de composants FLOSS (*free/libre open source software*). L'analyse de l'étude de cas permet de déterminer les conséquences négatives d'une documentation inadéquate sur le flux de connaissances au sein d'un projet de développement logiciel.

Le troisième article vise à comprendre les mécanismes menant aux divergences observées entre la conception et l'implémentation d'un projet de développement logiciel. L'utilisation de

la méthodologie ATS et du modèle de flux de connaissances facilite l'analyse de l'étude de cas. Ainsi, les discordances entre les artefacts de conception et l'implémentation s'expliquent par le fait que la conception n'est qu'une image de possibilités.

Les résultats de recherche complémentaires permettent de caractériser trois projets intégrateurs de développement logiciel, à la manière d'une étude de cas multiples de type exploratoire reposant sur la méthodologie ATS et la modélisation par flux de connaissances. Ainsi, l'analyse des jetons d'activité (ATS) permet de porter un jugement sur la rigueur des développeurs et donc sur la fiabilité des jetons, selon les trois profils identifiés. De plus, les facteurs cognitifs sont caractérisés selon leur caractère individuel et participatif. Par ailleurs, le séquencement cognitif permet l'identification de quatre profils de développeurs: le cristallisateur, le codeur, le polyvalent et l'agent libre. Finalement, une forte corrélation a été observée entre un effort d'acquisition élevé et une productivité du code source faible, ce qui constitue une contribution majeure, de par son originalité et ses conséquences théoriques et pratiques.

Les contributions de cette thèse sont de trois ordres: méthodologiques, théoriques et pratiques. Les contributions méthodologiques sont relatives à la méthodologie ATS. Le principal avantage de la méthodologie ATS est qu'elle permet l'analyse du développement logiciel sous une perspective différente de ce qui est possible avec les autres méthodologies utilisées en développement logiciel. De plus, la méthodologie offre l'avantage supplémentaire de sensibiliser les développeurs à ce qu'ils font, dans le cadre de projets intégrateurs. Parmi les contributions théoriques, deux ressortent du lot, soit le développement d'un modèle de flux de connaissances et la forte corrélation observée entre un effort d'acquisition élevé et une productivité du code source faible. Finalement, étant donné que les expérimentations ont été faites dans le cadre de projets intégrateurs, les contributions pratiques permettront d'abord et avant tout d'améliorer ces projets sous quatre aspects: la formation des équipes, le choix du projet, le choix du processus et la supervision des équipes.

La principale limitation de cette thèse est au niveau de sa validité externe. En effet, l'expérimentation étant basée sur des projets intégrateurs développés par des étudiants, il est légitime de se questionner sur la validité des résultats dans d'autres conditions, notamment en milieu industriel. Face à cette limitation potentielle, il est recommandé de conduire les mêmes

expérimentations dans un contexte industriel, de manière à prouver la validité externe des résultats.

Une avenue de recherche recommandée concerne l'extension de la méthodologie. En effet, il a été démontré que la méthodologie ATS permet d'analyser le développement logiciel dans une perspective de flux de connaissances et il serait très intéressant d'observer la symbiose de cet aspect avec d'autres problématiques complémentaires telles que les interactions ad hoc au sein d'une équipe de développement logiciel et la nature des interactions entre un développeur et son ordinateur, au cours du développement logiciel.

ABSTRACT

Given the failures faced by the software development industry and the inability of different software process approaches to solve these problems, it is interesting to rely, on the one hand, on achievements in the software process field and, on the other hand, learn from innovations in related fields. In particular, knowledge management applied to software engineering is a subject currently emerging. A better understanding of knowledge creation and conversion's mechanisms in software development projects is promising. The main objective of this thesis is to characterize software development projects from a knowledge flow perspective.

This doctoral thesis intends to achieve the research goal by presenting three papers and additional complementary results.

The first paper presents and justifies the methodology used in this thesis. The paper details the ATS (activity time slip) methodology, where software developers must log their activities from a knowledge perspective. Data are then codified based on the knowledge flow model, which is related to Nonaka and Takeuchi's (1995) knowledge creation model and which defines six cognitive factors: acquisition, crystallization, validation, implementation, verification and work organization. A multiple case study is presented to demonstrate the originality and relevance of the proposed methodology.

The second paper presents a case study using the knowledge flow model to analyze the consequences of documentation quality in FLOSS components reuse. The case study's analysis allows the identification of negative consequences on a software development project's knowledge flow resulting from inadequate documentation.

The third paper focuses on understanding the mechanisms leading to discrepancies between design and implementation in a software development project. The use of the ATS methodology and the knowledge flow model facilitates the case study's analysis. The discrepancies between design artifacts and implementation can be explained by the fact that design is an image of possibilities.

Complementary research results allow the characterization of three software development capstone projects by means of an exploratory multiple case study based on the ATS

methodology and the knowledge flow model. The analysis of the activity time slips (ATS) allows to judge the developers' rigor and therefore the reliability of their activity time slips, according to three identified patterns. Also, cognitive factors are characterized from a individual and participative viewpoint. Furthermore, cognitive sequencing allows the identification of four developers profiles: the crystallizer, the coder, the "versatile", and the free agent. Moreover, a strong correlation was observed between high acquisition effort and low source code productivity, which is a major contribution, because of its theoretical and practical implications.

Contributions of this thesis are threefold: methodological, theoretical and practical. The methodological contributions are related to the ATS methodology. Its main advantage is allowing software development's analysis from a different perspective of what is possible from other software development methodologies. In addition, in the context of capstone projects, the methodology educates developers on what they are doing. Among theoretical contributions, two stand out: the development of a knowledge flow model and the strong correlation between high acquisition effort and low source code productivity. Moreover, because the experiments were made in the context of capstone projects, practical contributions will first and foremost allow to enhance these projects in four areas: team creation, project selection, software process selection, and teams supervision.

The main limitation of this thesis is its external validity. Since experimentation is based on capstone projects developed by students, it is legitimate to question the validity of results in other conditions, especially in industrial setting. Given this potential limitation, it is recommended to conduct the same experiments in an industrial setting in order to prove the external validity of results.

Further research should focus on the methodology's extension. Since the ATS methodology allows software project analysis based on a knowledge flow perspective, it would be interesting to examine this aspect's symbiosis with other complementary issues such as ad hoc interactions within a software development team and the nature of interactions between a developer and his computer during software development.

TABLE DES MATIÈRES

REMERCIEMENTS iii

RÉSUMÉ iv

ABSTRACT vii

TABLE DES MATIÈRES ix

LISTE DES TABLEAUX xiii

LISTE DES FIGURES xiv

LISTE DES SIGLES ET ABRÉVIATIONS xvi

LISTE DES ANNEXES xvii

INTRODUCTION 1

CHAPITRE 1 REVUE CRITIQUE DE LA LITTÉRATURE 4

1.1 Processus logiciel 4

1.1.1 Processus basés sur l'ingénierie 5

1.1.2 Méthodes agiles 8

1.2 Amélioration de processus logiciels 11

1.2.1 Approche traditionnelle 11

1.2.2 Approche par recette 13

1.3 Perspective de connaissances 14

1.3.1 Données, informations et connaissances 14

1.3.2 Modèles de connaissances 14

1.3.3 Gestion des connaissances et processus logiciels 16

1.4 Synthèse de la littérature 18

CHAPITRE 2 DÉMARCHE DE L'ENSEMBLE DU TRAVAIL DE RECHERCHE ET ORGANISATION GÉNÉRALE DU DOCUMENT 19

CHAPITRE 3 A QUALITATIVE AND QUANTITATIVE DATA COLLECTION METHODOLOGY FOR KNOWLEDGE ANALYSIS IN SOFTWARE ENGINEERING 21

3.1 Abstract 21

3.2 Introduction 21

3.3 Methodology 23

3.3.1 Case study 23

3.3.2 Data collection techniques 25

3.3.3 Research data 29

3.3.4 Ethical considerations 29

3.4 Knowledge Perspective 30

3.4.1 Knowledge models 30

3.4.2 Knowledge management 31

3.5 Knowledge Flow in Software Projects 32

3.6 Discussion 40

3.6.1 Methodological challenges 40

3.6.2 Knowledge flow results 43

3.7 Conclusions 43

CHAPITRE 4 CONSEQUENCES OF DOCUMENTATION QUALITY IN FLOSS REUSE: A CASE STUDY 44

4.1 Abstract 44

4.2 Introduction 44

4.3 Methodological Approach 45

4.3.1 Capstone Project 45

4.3.2 Knowledge Flow Perspective 46

4.3.3 Effort Time Slip Method 48
4.3.4 Independent Data Codification 49
4.4 FLOSS Component Reuse: The Case of the SFLphone Capstone Project 50
4.4.1 Project Context 50
4.4.2 Disciplined Software Process 51
4.5 Analysis and Results 52
4.5.1 Time Slip Tokens 52
4.5.2 Knowledge Flow Analysis 53
4.5.3 FLOSS Library Issues 54
4.5.4 Consequences of Ambiguous Library Documentation 57
4.6 Discussion 58
4.6.1 Extra Effort Distribution 59
4.6.2 Software Practice Recommendation 61
4.7 Conclusions 62
4.8 Acknowledgments 63
CHAPITRE 5 IS DESIGN USEFUL IN SMALL SOFTWARE PROJECTS? AN EXPLORATORY CASE STUDY 64
5.1 Abstract 64
5.2 Introduction 64
5.3 Description of the Project 67
5.3.1 Class Categories 68
5.4 Design Process Activities 72
5.5 Discussion 74
5.6 Conclusion 78
5.7 Acknowledgments 78

CHAPITRE 6 RÉSULTATS COMPLÉMENTAIRES 80
6.1 Introduction 80
6.2 Modèle de flux de connaissances 80
6.3 Caractéristiques des projets analysés 83
6.4 Caractérisation des développeurs 85
6.5 Caractérisation de l'effort 90
6.5.1 Effort global 90
6.5.2 Travail individuel et participatif 92
6.5.3 Séquencement cognitif 102
6.5.4 Relation avec le code source 111
6.6 Discussion 112
CHAPITRE 7 DISCUSSION GÉNÉRALE 115
7.1 Contributions méthodologiques 115
7.2 Contributions théoriques 117
7.3 Contributions pratiques 117
7.3.1 Formation des équipes 117
7.3.2 Choix du projet 118
7.3.3 Choix du processus 118
7.3.4 Supervision des équipes 119
CONCLUSION ET RECOMMANDATIONS 120
LISTE DES RÉFÉRENCES 122
ANNEXES 136

LISTE DES TABLEAUX

Tableau 1.1: Processus SECI 15

Table 3.1: Activity Time Slip (ATS) Token Content 28

Table 3.2: Token examples 34

Table 3.3: Software development effort and tokens 35

Table 3.5: Project reliability index 42

Table 4.1: Time Slip Token Content 48

Table 4.2: Capstone Project Process 52

Table 4.3: Component effort distribution 58

Tableau 6.1 : Mots-clefs des facteurs cognitifs 83

Tableau 6.2 : Caractéristiques du projet C6 84

Tableau 6.3 : Caractéristiques du projet C7 84

Tableau 6.4 : Caractéristiques du projet C8 85

Tableau 6.5 : Caractéristiques des jetons individuels du projet C6 86

Tableau 6.6 : Caractéristiques des jetons individuels du projet C7 86

Tableau 6.7 : Caractéristiques des jetons individuels du projet C8 86

Tableau 6.8 : Caractéristiques des jetons individuels du projet C8 90

Tableau 6.9 : Effort investi par facteur cognitif pour le projet C6 90

Tableau 6.10: Effort investi par facteur cognitif pour le projet C7 91

Tableau 6.11 : Effort investi par facteur cognitif pour le projet C8 91

Tableau 6.12 : Répartition de l'effort individuel et participatif des projets C6 à C8 92

Tableau 6.13 : Profils des développeurs 105

Tableau 6.14 : Réalisation de code source et effort d'acquisition 111

LISTE DES FIGURES

Figure 1.1 Vue d'ensemble du RUP....6

Figure 1.2 Modèle bidimensionnel du UPEDU....7

Figure 3.1: Knowledge Flow Model....33

Figure 3.2: Project P06 total effort distribution....36

Figure 3.3: Project P07 total effort distribution....38

Figure 3.4: Project P08 total effort distribution....38

Figure 3.5: Project P09 total effort distribution....39

Figure 4.1: Knowledge Flow Model....47

Figure 4.2: Generic Process Practice....51

Figure 4.3: Cognitive Factor Effort Distribution in the SFLphone Project....54

Figure 4.4: Real and Adjusted Acquisition Cognitive Factor Cumulative Effort....60

Figure 4.5: Real and Adjusted Verification Cognitive Factor Cumulative Effort....60

Figure 4.6: Real and Adjusted Realization Cognitive Factor Cumulative Effort....61

Figure 4.7: Reusable Code Validation Practice....62

Figure 5.1: Model of class origin....69

Figure 5.2: Size in number of executable statements of the implemented classes....70

Figure 5.3: Size in number of executable statements of the designed classes that have been added and adapted....70

Figure 5.4: Product perspectives in terms of executable statements and number of classes for the designed classes and the classes not designed....71

Figure 5.5: Knowledge flow model....73

Figure 5.6: Model of cognitive activities performed during the design process....74

Figure 6.1 : Modèle du flux de connaissances d'un développeur logiciel....82

Figure 6.2 : Distribution de la durée des jetons individuels du projet C6....87

Figure 6.3: Distribution de la durée des jetons individuels du projet C7 87

Figure 6.4 : Distribution de la durée des jetons individuels du projet C8 88

Figure 6.5 : Profils de distribution de jetons α, β et γ 89

Figure 6.6 : Évolution de l'effort individuel et participatif du projet C6 93

Figure 6.7 : Évolution de l'effort individuel et participatif du projet C7 94

Figure 6.8 : Évolution de l'effort individuel et participatif du projet C8 94

Figure 6.9 : Évolution de l'effort d'acquisition individuel et participatif des projets C6 à C8 95

Figure 6.10 : Évolution de l'effort de cristallisation individuel et participatif des projets C6 à C8 96

Figure 6.11: Évolution de l'effort de validation individuel et participatif des projets C6 à C8 97

Figure 6.12 : Évolution de l'effort de réalisation individuel et participatif des projets C6 à C8 98

Figure 6.13 : Évolution de l'effort de vérification individuel et participatif des projets C6 à C8 99

Figure 6.14 : Évolution de l'effort d'organisation du travail individuel et participatif des projets C6 à C8 100

Figure 6.15 : Vue A-CV-RV du séquencement cognitif du développeur C7A 103

Figure 6.16 : Vue A-CR-VV du séquencement cognitif du développeur C7A 103

Figure 6.17 : Vue partielle A-CV-RV du séquencement cognitif du développeur C7A 104

Figure 6.18 : Vue partielle A-CR-VV du séquencement cognitif du développeur C7A 104

Figure 6.19 : Vue A-CV-RV du séquencement cognitif du développeur C8E 106

Figure 6.20 : Vue A-CR-VV du séquencement cognitif du développeur C8E 107

Figure 6.21 : Vue A-CV-RV du séquencement cognitif du développeur C7C 107

Figure 6.22 : Vue A-CR-VV du séquencement cognitif du développeur C7C 108

Figure 6.23 : Vue A-CV-RV du séquencement cognitif du développeur C6E 109

Figure 6.24 : Vue A-CR-VV du séquencement cognitif du développeur C6E 109

Figure 6.25 : Corrélation entre la réalisation de code source et l'effort d'acquisition 111

LISTE DES SIGLES ET ABRÉVIATIONS

ASD	*Adaptive Software Development*
ATS	*Activity Time Slip* (jeton d'activité), fait référence au même concept que ETS
CMMI	*Capability Maturity Model Integration*
DSDM	*Dynamic Systems Development Method*
ETS	*Effort Time Slip* (jeton d'effort), fait référence au même concept que ATS
FDD	*Feature-Driven Development*
FLOSS	*Free/libre open source software*
IDE	*Integrated Development Environment*
ISO/IEC	*International Organization for Standardization* et *International Electrotechnical Commission*
LD	*Lean Development*
MBASE	*Model-Based Architecting and Software Engineering*
RUP	*Rational Unified Process*
SECI	Socialisation, externalisation, combinaison et internalisation
SPICE	*Software Process Improvement and Capability dEtermination*
SW-CMM	*Software Capability Maturity Model*
UPEDU	*Unified Process for Education*
XP	*Extreme Programming*

LISTE DES ANNEXES

Annexe A Knowledge Conversion in Software Development.....136

Annexe B Exploring Knowledge Flow in Software Project Development.....141

Annexe C Échantillon type de jetons ATS.....148

INTRODUCTION

La dénomination « génie logiciel » est née de la Conférence du génie logiciel de l'OTAN tenue en 1968 pour répondre au besoin de mieux définir et encadrer les pratiques relatives au développement de systèmes logiciels (Naur & Randell, 1969). Malgré les diverses innovations au cours des années subséquentes, près de trois décennies plus tard, le *Standish Group* (1994) a publié le dévastateur rapport Chaos qui conclut que plus de la moitié (53%) des quelque 175 000 projets en technologie de l'information entrepris chaque année aux États-Unis n'atteignent pas les objectifs fixés (d'échéancier, de coût et de qualité), qu'environ le tiers (31%) sont tout simplement annulés avant d'être complétés et qu'uniquement le sixième (16%) des projets atteint les objectifs fixés. Face à cette situation alarmante, la réaction de l'industrie a été plus musclée que par le passé. Notamment, la gestion de projet et la gestion des exigences ont été renforcées. De plus, les processus basés sur l'ingénierie ont été popularisés. Toutefois, dans sa mise à jour du rapport Chaos en 2009, le Standish Group rapporte que, comparativement à 1994, les projets n'atteignant pas les objectifs sont en baisse de 9% (44%), les projets annulés sont en baisse de 7% (24%), alors que les projets atteignant les objectifs fixés sont en hausse de 16% (32%) (Eveleens & Verhoef, 2010). Force est de constater que, malgré une amélioration considérable de la situation en 15 ans, beaucoup de progrès reste à faire.

Au fil des années, plusieurs modèles de développement logiciel on été élaborés. Or, qu'on pense au modèle en cascades, au modèle en spirale ou un modèle itératif, force est de constater que le développement logiciel est difficile à modéliser. Des années 1970 à 2000, les processus développés étaient principalement prédictifs et basés sur la production d'artefacts, souvent dans le but de satisfaire des normes. À l'opposé, principalement depuis le début du XXI^{e} siècle, les méthodologies agiles, qui sont réactives et qui mettent l'accent sur les ressources humaines, gagnent en popularité. Toutefois, ce mouvement soulève le scepticisme dans les milieux où la traçabilité est essentielle (processus fortement basé sur les normes, systèmes critiques, etc.).

Face à la problématique des différentes approches de processus logiciels, il s'avère intéressant de se baser, d'une part, sur les acquis du domaine des processus logiciels et, d'autre part, de s'inspirer des innovations de domaines connexes. En particulier, la gestion des connaissances appliquée au génie logiciel est un sujet présentement en émergence. Une meilleure

compréhension des mécanismes de création et de conversion de connaissances au sein d'un projet de développement logiciel est une avenue de recherche prometteuse.

L'objectif principal de cette thèse de doctorat est donc de caractériser les projets de développement logiciel dans une perspective de flux de connaissances. L'objectif sera atteint par la présentation de trois articles complémentaires en plus d'un chapitre détaillant des résultats complémentaires.

Le chapitre 1 présente une revue de la littérature relative au contexte de l'objectif de recherche. Ainsi, les différents types de processus logiciels sont détaillés, les concepts d'amélioration de processus logiciels sont expliqués et la perspective de connaissances est étudiée.

Le chapitre 2 détaille la démarche de l'ensemble du travail de recherche et l'organisation générale du document.

Les chapitres 3 à 5 présentent 3 articles soumis pour publication à des revues avec comité de lecture.

Plus précisément, le chapitre 3 introduit le premier article de revue. Cet article a pour principal objectif la présentation et la justification de la méthodologie utilisée dans le cadre des travaux de recherche présentés dans cette thèse.

Le chapitre 4 présente le second article de revue. Cet article expose une étude de cas permettant d'analyser les conséquences de la qualité de la documentation lors de la réutilisation de composants FLOSS (*free/libre open source software*).

Le chapitre 5 introduit le troisième et dernier article de revue. Cet article a pour but de comprendre les mécanismes menant aux divergences observées entre la conception et l'implémentation d'un projet de développement logiciel.

Le chapitre 6 présente des résultats de recherche complémentaires, pas encore soumis pour publication. L'objectif du chapitre est de caractériser 3 projets intégrateurs, à la manière d'une étude de cas multiples de type exploratoire.

Le chapitre 7 propose une discussion générale des apports méthodologiques, théoriques et pratiques de l'ensemble des travaux de recherche présentés dans ce document.

En annexe se trouvent deux articles de conférences publiés antérieurement à la soumission des articles de revue. Le premier vise à établir les bases du sujet de recherche. Le second vise à introduire le modèle de flux de connaissances, soit la contribution théorique principale de cette thèse de doctorat.

CHAPITRE 1

REVUE CRITIQUE DE LA LITTÉRATURE

Ce chapitre vise à présenter la littérature relative à l'objectif de recherche, soit la caractérisation de projet de développement logiciel dans une perspective de flux de connaissances. Or, la littérature précisément reliée à l'objectif de recherche étant très limitée, quelques éléments théoriques pertinents sont présentés.

Dans un premier temps, les différents types de processus logiciels sont détaillés (section 1.1). Par la suite, les concepts d'amélioration de processus logiciels sont expliqués (section 1.2). Puis, la perspective de connaissances est étudiée (section 1.3). Finalement, une synthèse des écrits (section 1.4) conclut ce chapitre.

1.1 Processus logiciel

Depuis environ une décennie, deux types d'approche de développement logiciel se démarquent soit celle dite traditionnelle (Kettunen & Laanti, 2005; Nerur, Mahapatra, & Mangalaraj, 2005), ou basée sur l'ingénierie (Fowler, 2005; Germain & Robillard, 2005) et l'autre dite agile (Cockburn, 2002; Fowler, 2005) ou légère (Zettel, Maurer, Munch, & Wong, 2001; Nawrocki, Walter, & Wojciechowski, 2002).

La première soutient que le développement logiciel doit être un processus discipliné incorporant des notions de mathématiques, de science et d'ingénierie (Bailetti & Liu, 2003). La seconde suggère que le développement logiciel est un processus créatif et agile (Rifkin, 2001).

Certains croient que la tendance actuelle en développement logiciel est de délaisser les imposants processus organisationnels rigides au profit des processus agiles et adaptables (Kettunen & Laanti, 2005).

Neill (2003) croit que les méthodes agiles dépendent grandement du talent des développeurs, ce qui constitue un risque considérable pour une organisation. Les méthodes agiles préconisent les individus et les interactions plutôt que les processus et les outils, un logiciel fonctionnel plutôt

qu'une documentation efficace, la collaboration du client plutôt que la négociation de contrat, ainsi que la réponse au changement plutôt que la poursuite d'un plan. Toutefois, l'auteur met en garde contre l'absence d'universalité des méthodes agiles en particulier pour les applications critiques et les grosses organisations.

Pour sa part, Fowler (2005) résume la différence fondamentale des méthodes agiles par rapport aux processus traditionnels à deux caractéristiques: elles sont adaptatives plutôt que prédictives et elles sont orientées ressources humaines plutôt qu'orientées processus.

Par ailleurs, il importe de souligner l'existence de plusieurs processus propriétaires, centrés sur la gestion, qui sont utilisés dans certains grands projets de développement informatique (gouvernements, banques, etc.). Ces processus, dont Macroscope (Fujitsu, 2006) est un exemple, sont en fait des adaptations des méthodes génériques discutées dans la littérature scientifique et ne seront donc pas considérés dans le cadre de cette thèse.

1.1.1 Processus basés sur l'ingénierie

Les processus basés sur l'ingénierie visent la production d'artefacts pour supporter la prise de décision concernant les exigences et la conception; le principe de base étant que les efforts d'activités de planification de production d'artefacts résulteront en un coût global plus bas, une livraison de produit à temps et une meilleure qualité logicielle (Germain & Robillard, 2005).

Les processus traditionnels sont surtout utilisés au sein d'entreprises devant respecter des normes pour des raisons d'affaires. D'ailleurs, ces organisations se forgent souvent un processus à partir des normes auxquelles elles doivent se conformer. Il existe aussi des modèles de processus à partir desquels il est possible d'adapter un processus selon les besoins de l'organisation, dont le *Rational Unified Process*¸ le *Unified Process for Education* et *Model-Based Architecting and Software Engineering*.

1.1.1.1 Rational Unified Process

Le *Rational Unified Process* (RUP) (Kruchten, 2000) est un modèle de processus basé sur les cas d'utilisation, centré sur l'architecture, itératif et incrémental, développé par *Rational Software Corporation* (maintenant IBM). L'objectif du RUP est d'assurer le développement de logiciel de haute qualité qui respecte les besoins des utilisateurs, l'échéancier et le budget. Plus précisément,

le RUP est un cadre de référence (*framework*) d'un cycle de développement logiciel. Il oriente l'équipe de développement autant pour les activités de gestion que pour les activités techniques.

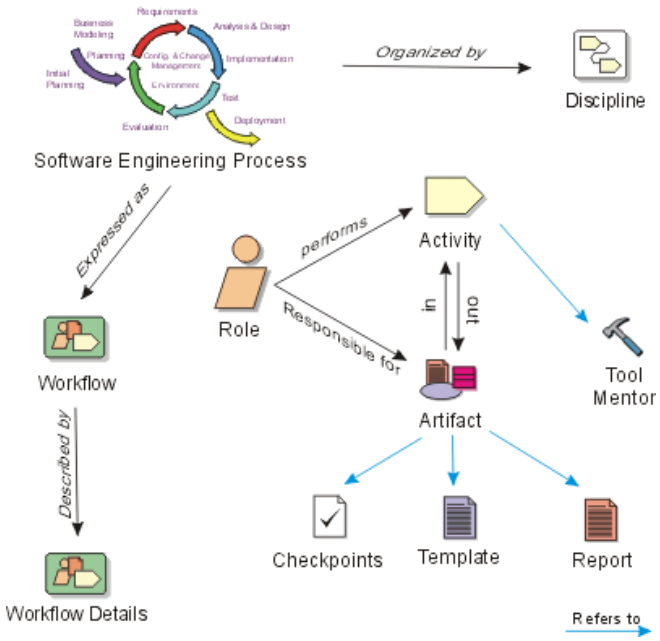

Figure 1.1 Vue d'ensemble du RUP
(Tiré de RUP)

La Figure 1.1 présente une vue d'ensemble des éléments de base du RUP. D'abord, le processus logiciel est organisé par disciplines, qui regroupent des activités de même type. Puis, ces disciplines sont exprimées par des flux de travail (*workflow*), qui sont des séquences d'activités produisant des résultats observables. À leur tour, ces flux de travail sont décrits par des détails de flux de travail qui sont représentés par des activités faites par des rôles, en utilisant des outils, dans le but de générer des artefacts. Pour leur part, les activités sont des unités de travail décrites en étapes concrètes. En ce qui a trait aux rôles, ils définissent le comportement et les responsabilités d'individus ou de groupes d'individus dans le contexte d'une organisation logicielle. Finalement, un artefact est un produit de travail issu du processus logiciel.

Les disciplines sont des ensembles d'activités reliées à un type de problème au sein d'un projet. Le RUP en possède neuf soit la modélisation d'affaires, les exigences, l'analyse et conception, l'implémentation, les tests, le déploiement, la gestion de la configuration et des changements, la gestion de projet et l'environnement.

1.1.1.2 Unified Process for Education

Le *Unified Process for Education* (UPEDU) (Robillard, Kruchten, & d'Astous, 2003) est un modèle de processus dérivé du RUP. La particularité du UPEDU est qu'il a été adapté au domaine académique. Par exemple, trois des neuf disciplines ont été supprimées du RUP parce qu'elles s'appliquaient mal au contexte académique.

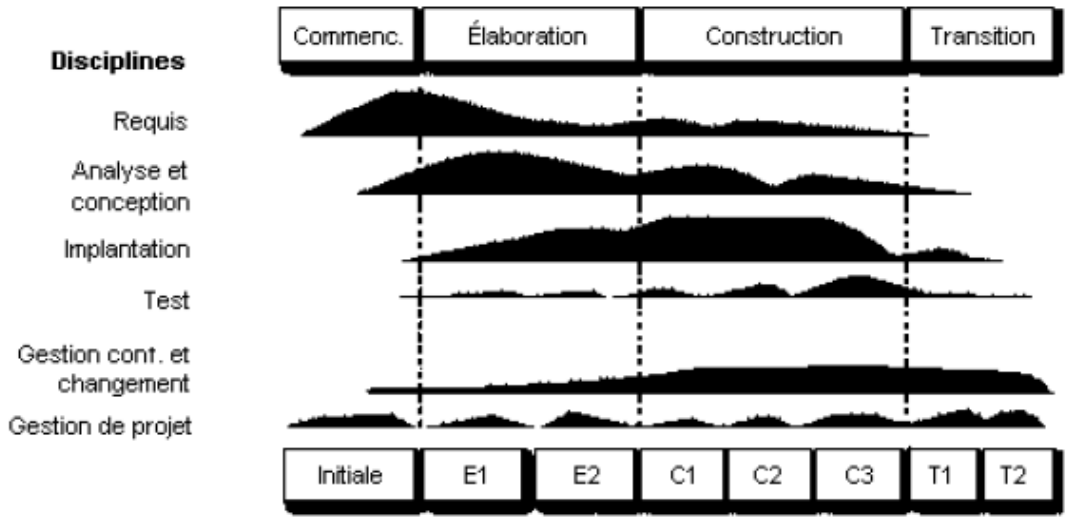

Figure 1.2 Modèle bidimensionnel du UPEDU
(Tiré de Germain (2004))

La figure 1.2 explicite le modèle bidimensionnel de processus et de cycle de vie du UPEDU. Les quatre phases d'un cycle de développement sont situées en abscisse et sont divisées en huit itérations, tandis que les six disciplines se trouvent en ordonnées. Pour leur part, les six courbes représentent une possibilité de répartition de l'effort au sein de disciplines selon l'itération.

1.1.1.3 Model-Based Architecting and Software Engineering

Le *Model-Based Architecting and Software Engineering* (MBASE) (Boehm, Port, Egyed, & Abi-Antoun, 1999) est un modèle de processus qui se veut une extension du RUP axée sur l'ingénierie système utilisant intensément le logiciel (*software-intensive system engineering*). À ce sujet, les quatre caractéristiques principales sont l'évitement de conflit de modèles (*model clash avoidance*); l'intégration de modèles et le cadre de processus (*framework process*); les

négociations gagnant-gagnant des exigences des intervenants; les jalons de point d'ancrage de cycle de vie et les critères de réussite-échec (Boehm, Port, & Basili, 2002).

En somme, MBASE est une autre mouture de processus basé sur l'ingénierie, avec la particularité d'être axée sur l'ingénierie système.

1.1.2 Méthodes agiles

Les méthodes agiles s'inscrivent en opposition aux méthodes traditionnelles qui sont mal adaptées trop lourdes pour répondre assez rapidement aux fréquents changements d'environnement (Erickson, Lyytinen, & Keng, 2005). À ce propos, un manifeste pour le développement agile de logiciels (*Agile Manifesto*) a été développé par les promoteurs et leaders des méthodes agiles (Beck, K., et al., 2001).

Afin de parvenir à s'adapter aux changements d'environnement, les méthodes agiles divisent le projet en sous-projets fonctionnels réalisés en incréments relativement courts (3 à 12 semaines) et mettent l'accent sur la gestion des connaissances tacites au lieu de la documentation externe (Cockburn, 2002).

Les méthodes agiles possèdent toutes les caractéristiques suivantes : itératives, incrémentales, autoorganisées et émergentes (Lindvall, M., et al., 2002).

Il existe un grand nombre de méthodes agiles dont les plus importantes sont : *Extreme Programming*, Scrum, *Dynamic Systems Development Method*, *Adaptive Software Development*, *Crystal*, *Lean Development* et *Feature-Driven Development*.

1.1.2.1 Extreme Programming

Certainement la plus populaire des méthodes agiles, le *Extreme Programming* (XP) est destiné aux équipes de petite ou moyenne taille évoluant dans un environnement de développement où les exigences sont vagues ou changent rapidement (Beck, 1999a).

XP est constitué de quatre valeurs – communication, simplicité, rétroaction et courage – et de douze pratiques : jeu de planification, courts délais de livraison, métaphores, conception simple, tests, refactorisation, programmation par paire, intégration continue, propriété collective, client sur le site, semaines de 40 heures et espace de travail ouvert (Beck, 1999b).

XP propose un cycle incrémental de développement qui suit des itérations très courtes (quelques semaines), dans le but de tirer parti du coût du changement d'un logiciel au cours du cycle de vie. Une itération typique est déterminée par le client qui choisit les caractéristiques, sous forme scénarios utilisateurs (*user stories*) qu'il désire voir implémentées, selon leur valeur et leur coût. Par la suite, les histoires sont divisées en tâches, qui sont distribuées aux programmeurs par équipe de deux (programmation par paire, aussi appelée programmation par binôme). Pour chaque tâche, les tests unitaires sont implémentés et exécutés préalablement à l'implémentation de la tâche, de manière à constituer une base automatique de tests du système. Ce type de développement basé sur les tests donne aux développeurs, au fil du temps, confiance au comportement de leur système (Beck, 1999a).

1.1.2.2 Scrum

Scrum est une méthode agile pour la gestion de projet, qui préconise la construction par incrément pour les environnements complexes au sein d'équipes de moins de 10 développeurs (Schwaber & Beedle, 2002). Les *sprints*, qui s'étendent sur une à quatre semaines, constituent les itérations de développement. Chaque sprint possède une date fixe de livraison et le produit livré constitue un incrément par rapport au sprint précédent. Les tâches à faire au cours d'un sprint sont consignées dans le *backlog* et sont réparties au sein de l'équipe, qui est dirigée par le *scrum master* (Rising & Janoff, 2000). Scrum est principalement axé sur la gestion de projet, notamment par le développement itératif et le contrôle accru à l'aide de réunions quotidiennes (appelées scrum) (Fowler, 2005).

1.1.2.3 Dynamic Systems Development Method

Le *Dynamic Systems Development Method* (DSDM) (Stapleton, 1997) se concentre sur une modélisation holistique du développement logiciel en mettant l'accent sur les boucles de rétroaction, donc sur la gestion des connaissances. L'essentiel de l'argumentation repose sur une vision du développement caractérisée par des systèmes intimement reliés à leur environnement et ayant tendance à évoluer de pair avec ceux-ci. Le modèle correspond à un ensemble de boucles d'interactions et de rétroactions contrôlant la production de logiciel sous une perspective continue et dynamique. Une telle modélisation est dynamique, rétroactive, centrée sur la gestion de risques, multidimensionnelle, continue, en plus d'inclure la connaissance et l'incertitude

(Dalcher, 2003). Or, bien que la modélisation du DSDM soit impressionnante, le cadre très théorique rend son application complexe.

1.1.2.4 Adaptive Software Development

Highsmith (1997) a développé le *Adaptive Software Development* (ASD) afin de faire face au développement de logiciels complexes imprédictibles et non linéaires. Le concept clef de cette théorie est l'émergence comme réponse à l'absence de déterminisme, donc à l'impossibilité de déterminer le lien entre cause et effet. En somme, ASD est principalement centré sur la gestion du changement.

1.1.2.5 Crystal

Les méthodologies Crystal ont été développées par Cockburn comme un groupe d'approches adaptées à différentes tailles d'équipe et différents degrés de criticité de systèmes logiciels (Fowler, 2005).

Malgré leurs variations, toutes les approches Crystal partagent des caractéristiques communes dont les trois priorités sont la sûreté (de l'issue du projet), l'efficacité et l'habitabilité (possibilité pour les développeurs de coexister avec Crystal). Elles ont aussi des propriétés communes dont les trois plus importantes sont les livraisons fréquentes, l'amélioration réflective et une solide communication.

En somme, les méthodologies Crystal mettent l'emphase sur la planification et la gestion de projet afin de gérer notamment les communications déficientes au sein d'un processus de développement.

1.1.2.6 Lean Development

Le *Lean Development* (LD) est une stratégie provenant de la gestion de production qui vise à continuellement améliorer les processus d'affaires en mettant l'accent sur les activités générant de la valeur aux yeux du client (Poppendeick & Poppendeick, 2003). Le LD et le XP ont plusieurs points en commun: méthodologie utilisée, adaptation aux besoins changeants, travail d'équipe, itérations et contrôle de la qualité. À l'opposé, les deux approches possèdent des

différences : transfert de la connaissance, méthodes de mesures et répartition des coûts (Dall'Agnol, Janes, Succi, & Zaninotto, 2003).

En somme, le LD vise à réduire la complexité au cours du processus en retardant les décisions aux lourdes conséquences le plus tard possible.

1.1.2.7 Feature-Driven Development

Le *Feature-Driven Development* (FDD) (Palmer & Felsing, 2002) est centré sur les intervenants et l'architecture. Comme son nom l'indique, le FDD est basé sur les caractéristiques, de la même manière que le RUP est basé sur les cas d'utilisation. Le cycle de vie est composé de cinq étapes : développer un modèle global, produire une liste de caractéristiques, planifier par caractéristiques, concevoir par caractéristique et construire par caractéristique. Les itérations sont d'une durée de deux semaines ou moins.

En somme, le FDD concentre le développement logiciel sur les caractéristiques du produit.

1.2 Amélioration de processus logiciels

L'amélioration de processus logiciels (*software process improvement*) est née d'un mouvement pour la qualité initiée par Crosby (1979), Deming (1986) et Juran (1988).

Une organisation qui désire améliorer son processus logiciel peut utiliser deux types d'approche soit l'approche traditionnelle, dite par plan (*blueprint*) et l'approche par lignes directrices, dite par recette (*recipe*). L'approche traditionnelle met l'accent sur le formalisme et la conformité d'un processus à des modèles de référence, alors que l'approche par recette met l'accent sur les connaissances des utilisateurs du processus.

1.2.1 Approche traditionnelle

Les organisations désirant améliorer leur processus logiciel par l'approche traditionnelle tentent de se conformer à des modèles de référence de processus. Les quatre modèles de référence les plus connus sont ISO 9001, SW-CMM, CMMI et ISO/IEC 15504.

1.2.1.1 ISO 9001

ISO 9000 est une série de normes internationales conçues pour la gestion et l'assurance de la qualité qui spécifie les exigences de base pour le développement, la production, l'installation et la mise en service au niveau du système et au niveau du produit. En particulier, ISO 9001 (avec les lignes directrices de ISO 9000-3) est applicable au développement et à la maintenance de systèmes logiciels (Jung & Hunter, 2001). La détermination de la capacité d'une organisation se fait à partir d'une liste de points de contrôle et tous les points doivent être satisfaits afin que l'organisation en question soit considérée comme respectant la norme de qualité ISO 9000 (Wang, Y., et al., 1997).

Le principal problème avec ce modèle de référence est, d'une part, qu'il est mal adapté au domaine logiciel et, d'autre part, qu'il ne permet qu'une évaluation globale de la qualité.

1.2.1.2 Software Capability Maturity Model (SW-CMM)

Le SW-CMM (Paulk, Curtis, Chrissis, & Weber, 1993), développé par le Software Engineering Institute (SEI), catégorise un ensemble de pratiques clefs de développement logiciel en 18 secteurs clefs, qui sont eux-mêmes regroupés en 5 niveaux de capacité cumulatifs. L'atteinte d'un niveau de capacité est associée à un niveau de maturité de processus, ce qui survient lorsque tous les buts associés à tous les secteurs clefs d'un niveau sont respectés.

Le principal avantage de ce modèle est la simplicité de comparaison que procurent les 5 niveaux de capacité. En d'autres mots, deux organisations possédant un même niveau de maturité seront considérées comme ayant des processus de même qualité. Or, le caractère étagé (*staged*) du modèle est simpliste. Par exemple, une organisation qui respecte tous les secteurs clefs du niveau 2, sauf un but d'un secteur clef, sera considérée comme de niveau 1 (initial), soit le même niveau de maturité qu'une entreprise n'ayant qu'un processus aléatoire.

1.2.1.3 Capability Maturity Model Integration (CMMI)

Le *Capability Maturity Model Integration* (CMMI) est basé sur le CMM-SW, mais se distingue par le fait qu'il intègre quatre disciplines (au lieu d'une seule) soit l'ingénierie système (*system engineering*), l'ingénierie logicielle (*software engineering*), le développement intégré de produit

et processus (*integrated product and process development*) et l'approvisionnement (*supplier sourcing*).

De plus, le CMMI corrige le principal inconvénient du SW-CMM, en ajoutant la représentation continue (*continuous*). Ainsi, un niveau de capacité, sur une échelle de 0 à 5, est associé à chaque secteur clef. Le profil d'une organisation peut donc être déterminé selon le niveau de capacité de chaque secteur clef, au lieu d'être réduit à représentation étagée (*staged*).

1.2.1.4 ISO/IEC 15504

ISO/IEC 15504 (ISO, 2003) est une norme internationale aussi connue sous le nom de Software Process Improvement and Capability Determination (SPICE). Elle possède une architecture d'évaluation de processus à deux dimensions : les processus et la capacité. Dans la dimension processus, les différents processus (au sens d'ensemble de pratiques), qui sont associés au développement et à la maintenance du logiciel, sont divisés en cinq catégories soit client-fournisseur, ingénierie, support, gestion et organisation. Pour sa part, la dimension capacité est représentée par des attributs de processus (PA) et est divisée en 6 niveaux.

SPICE remédie aussi au principal inconvénient de SW-CMM en déterminant la capacité propre à chacune des cinq catégories de processus. Ainsi, les résultats de l'évaluation sont plus détaillés pour SPICE que pour SW-CMM.

1.2.2 Approche par recette

Un défi important de l'amélioration de processus est de s'assurer que les utilisateurs de processus logiciels partagent une compréhension commune du processus. L'amélioration de processus nécessite le transfert et la construction de connaissances de processus des individus et de l'organisation, ce que l'on peut traduire par une problématique de gestion des connaissances. Souvent, l'amélioration de processus traditionnelle met l'accent sur la description et la prescription au détriment de la compétence et du comportement.

L'approche par recette propose des lignes directrices insistant sur le rôle central des connaissances tacites dans le partage et la création d'informations, ainsi que sur l'importance de la modélisation de ce que les utilisateurs de processus font (processus réel) plutôt que ce qu'ils devraient faire (processus prescrit) (Aaen, 2003).

1.3 Perspective de connaissances

Il y a déjà deux décennies, Alvin Toffler (1990) prédisait l'imminence d'une société basée sur les connaissances comme source de pouvoir. En ce début de XXI^{e} siècle, les connaissances s'avèrent, en effet, une arme stratégique cruciale pour les entreprises en quête de productivité accrue, d'où l'importance de la gestion des connaissances (Choi & Lee, 2002).

Les informations et les connaissances sont les forces vitales des organisations d'aujourd'hui (Trandsen & Vickery, 1998) et particulièrement des organisations logicielles. En effet, les connaissances sont primordiales au cours du cycle de développement d'un produit logiciel, particulièrement lors de la conception.

1.3.1 Données, informations et connaissances

D'entrée de jeu, il est important de bien faire la distinction entre données, informations et connaissances. En effet, les connaissances sont composées d'informations qui, elles, sont composées de données (Williams, 2006). De plus, la connaissance est spécifique au contexte, car elle dépend du temps et de l'espace. L'information devient connaissance lorsqu'elle est interprétée par un individu, associée à un contexte et ancrée dans les croyances et engagements d'un individu (Nonaka & Takeuchi, 1995).

Il existe deux types de connaissances : les connaissances explicites et les connaissances tacites (Polanyi, 1997). Les connaissances explicites peuvent être exprimées en langage formel et systématique. Elles peuvent être traitées, transmises et conservées relativement facilement (Williams, 2006). À l'opposé, les connaissances tacites sont hautement personnelles et difficiles à formaliser. Les connaissances tacites sont profondément ancrées dans les actions, procédures, routines, engagements, idéaux, valeurs et émotions d'individus (Schon, 1983).

1.3.2 Modèles de connaissances

En sciences cognitives, plusieurs modèles ont été développés afin de représenter les connaissances, mais quatre modèles sont particulièrement reconnus (Bjornson & Dingsoyr, 2008). Il s'agit du modèle d'apprentissage expérientiel de Kolb, de la théorie d'apprentissage par double boucle d'Argyris & Schon, de la théorie des communautés de pratique de Wenger et du modèle de création de connaissances de Nonaka & Takeuchi.

Kolb (1984) décrit l'apprentissage expérientiel par quatre modes d'apprentissage répartis dans deux dimensions. Une dimension réfère à l'appropriation d'expérience et inclut deux modes: la compréhension par conceptualisation abstraite et l'appréhension par expériences concrètes. L'autre dimension réfère à la conversion d'expérience et inclut également deux modes: l'intention par observation réflective et l'extension par expérimentation active. Selon Kolb, les quatre modes doivent être utilisés afin de maximiser l'apprentissage.

Argyris & Schön (1978) différentient l'apprentissage par simple et par double boucle. L'apprentissage par simple boucle implique la génération de nouvelles stratégies d'action sans modification de valeurs de gouvernance, alors que l'apprentissage par double boucle implique l'adaptation et la modification de stratégies et de valeurs de gouvernance.

Face au processus traditionnel d'apprentissage individuel, Wenger (1998) propose un processus d'apprentissage social qu'il appelle la communauté de pratique. Les membres d'une communauté de pratique s'impliquent dans un processus d'apprentissage collectif par le partage de connaissances issues d'une même pratique.

Selon Nonaka & Takeuchi (1995), la connaissance est créée par l'interaction entre les connaissances explicites et tacites. Ils proposent un modèle, le processus SECI, définissant quatre types de conversion de connaissances : socialisation, externalisation, combinaison et internalisation. Le tableau 1.1 présente les types de conversion impliqués selon les connaissances initiales et finales.

Tableau 1.1: Processus SECI

Connaissances initiales	**Connaissances finales**	**Type de conversion de connaissances**
Tacites	Tacites	Socialisation
Tacites	Explicites	Externalisation
Explicites	Explicites	Combinaison
Explicites	Tacites	Internalisation

La socialisation est le processus de conversion de nouvelles connaissances tacites lors d'expériences partagées. Elle survient typiquement dans le cadre de relation maître-apprenti où l'apprentissage des connaissances tacites se fait par l'expérimentation plutôt que par la lecture de manuels. Certaines organisations tirent d'ailleurs profit des connaissances tacites détenues par leurs fournisseurs et leurs clients en interagissant avec eux.

L'externalisation est le processus d'articulation des connaissances tacites en connaissances explicites. Lorsque les connaissances tacites sont explicitées, les connaissances sont cristallisées, leur permettant d'être partagées, devenant ainsi la base de nouvelles connaissances. La création de concepts dans le développement d'un nouveau produit en est un exemple.

La combinaison est le processus de conversion de connaissances explicites en d'autres connaissances explicites plus complexes ou systématiques. L'agrégation et la fragmentation de concepts font partie du processus de combinaison.

L'internalisation est le processus d'incorporation de connaissances explicites en connaissances tacites. La formation est l'exemple par excellence de l'internalisation, où des individus s'approprient au sein de leurs connaissances tacites des connaissances explicites de l'organisation.

1.3.3 Gestion des connaissances et processus logiciels

La gestion des connaissances est un vaste champ interdisciplinaire (Bjornson & Dingsoyr, 2008). Earl (2001) suggère une taxonomie de stratégies de gestion de connaissances, qu'il nomme des écoles (*schools*), selon trois catégories: technocratiques, économiques ou comportementales. Les écoles technocratiques incluent: l'école système, insistant sur le partage de connaissances; l'école cartographique, s'intéressant à la cartographie des connaissances organisationnelles; l'école d'ingénierie, mettant l'accent sur les processus et le flux de connaissances dans les organisations. L'école économique s'intéresse à l'exploitation commerciale des connaissances et du capital intellectuel. Les écoles comportementales incluent: l'école organisationnelle, se concentrant sur les réseaux pour le partage des connaissances; l'école spatiale, se concentrant sur la façon dont les bureaux peuvent être conçus afin de promouvoir le partage des connaissances; l'école stratégique, considérant la gestion de connaissances en tant qu'outil stratégique.

L'école d'ingénierie, se concentrant principalement sur le processus, est l'école de gestion de connaissances recevant la plus importante attention empirique (Bjornson & Dingsoyr, 2008). Deux catégories peuvent être identifiées au sein de cette école. La première s'intéresse au processus logiciel complet en ce qui concerne la gestion des connaissances. La seconde considère les possibilités d'amélioration d'activités spécifiques au sein d'un processus logiciel.

Par rapport à la première catégorie, Alavi & Leidner (2001) croient que le principal défi de gestion des connaissances est de faciliter le flux de connaissances entre les individus de manière à maximiser la quantité de connaissances transférées.

Arent & Norbjerg (2000) ont étudié l'amélioration de processus logiciels d'une perspective de connaissances basée sur le modèle de création de connaissances de Nonaka & Takeuchi. Ils ont conclu qu'autant les connaissances tacites que les connaissances explicites sont cruciales au succès de l'amélioration de processus logiciels. Les connaissances tacites sont nécessaires pour modifier les pratiques, alors que les connaissances explicites sont nécessaires afin de créer une mémoire organisationnelle.

Nerur & Balijepally (2007) affirment que le type de processus logiciel a un impact sur la manière de gérer les connaissances. L'approche traditionnelle repose essentiellement sur la gestion de connaissances explicites, tandis que les méthodes agiles se fondent principalement sur la gestion de connaissances tacites.

Dahkli & Chouikha (2009) suggèrent un processus de développement logiciel orienté connaissances conçu de manière à réduire l'écart entre les connaissances réellement intégrées dans les systèmes logiciels et les connaissances détenues par les acteurs organisationnels.

Par rapport à la seconde catégorie, Melnik & Maurer (2004) s'intéressent au rôle de la conversation et de l'interaction sociale en tant qu'éléments clés de l'efficacité du partage des connaissances dans un processus agile. Ils concluent que le partage des connaissances explicites est inefficace lorsque des artefacts cognitifs complexes sont utilisés. Plus le niveau de complexité est élevé, plus un partage interactif de connaissances est nécessaire, par le biais de communications verbales directes.

Bjornson & Dingsoyr (2005) ont étudié le partage des connaissances par le tutorat dans une petite entreprise de consultation dans le domaine logiciel. Afin d'améliorer le mentorat, ils proposent d'introduire des méthodes pour augmenter le niveau de réflexion des employés.

Desouza, Awazu, & Wan (2006) ont examiné les facteurs qui contribuent à l'utilisation de connaissances explicites dans une organisation de génie logiciel. Ils ont constaté que la complexité perçue, l'avantage relatif perçu, ainsi que les risques perçus sont les facteurs affectant les connaissances explicites.

En somme, la littérature relative à l'école d'ingénierie de la gestion de connaissances s'intéresse aux pratiques et au processus logiciels dans une optique de gestion de connaissance, mais pas spécifiquement dans une perspective empirique de flux de connaissances.

1.4 Synthèse de la littérature

Les processus logiciels sont principalement catégorisés selon deux types d'approches. D'une part, on retrouve les processus basés sur l'ingénierie tels que RUP, UPEDU et MBASE. D'autre part, se trouvent les processus agiles tels que XP, Scrum, DSDM, ASD, Crystal, LD et FDD.

Une organisation qui désire améliorer son processus logiciel peut utiliser deux types d'approche soit l'approche traditionnelle, dite par plan (*blueprint*) et l'approche par lignes directrices, dite par recette (*recipe*). À ce sujet, on remarque plusieurs similitudes entre d'une part, les processus traditionnels et l'amélioration de processus traditionnelle et, d'autre part, les processus agiles et l'amélioration de processus par recette.

Face à cette dualité entre les approches traditionnelles et agiles/par recette, il s'avère intéressant d'explorer d'autres alternatives en s'inspirant des innovations de domaines connexes, notamment la perspective de connaissances relative à la gestion de connaissances.

L'école d'ingénierie, se concentrant principalement sur le processus, est l'école de gestion de connaissances recevant la plus importante attention empirique. À ce sujet, le modèle de création de connaissances de Nonaka & Takeuchi étant le plus utilisé. Or, bien que la littérature s'intéresse aux pratiques et au processus logiciels dans une optique de gestion de connaissance, il existe un vide quant à la perspective empirique de flux de connaissances.

Conséquemment, l'objectif principal de cette thèse, étant de caractériser les projets de développement logiciel dans une perspective de flux de connaissances, est original et s'inscrit en continuité avec la l'état de la littérature.

CHAPITRE 2

DÉMARCHE DE L'ENSEMBLE DU TRAVAIL DE RECHERCHE ET ORGANISATION GÉNÉRALE DU DOCUMENT

Le type de présentation retenu pour ce document est la thèse par articles. Ainsi, les trois prochains chapitres présentent trois articles soumis à des revues avec comité de lecture. De plus, le chapitre 6 contient des résultats de recherche complémentaires, pas encore été soumis pour publication.

Par ailleurs, deux articles de conférences ont été publiés antérieurement à la soumission des articles de revue. Ces 2 articles se trouvent en annexe. D'abord, l'article intitulé "Knowledge Conversion in Software Development" a été présenté dans le cadre de la *Nineteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'2007)* à Boston, aux États-Unis, en juillet 2007. Cet article visait à établir les bases du sujet de recherche. Plus précisément, l'article suggère une perspective de connaissances, basé sur le processus SECI de Nonaka & Takeuchi, comme moyen d'analyse d'un projet de développement logiciel. Pour sa part, l'article intitulé "Exploring Knowledge Flow in Software Project Development" a été présenté dans le cadre de la *2009 International Conference on Information, Process, and Knowledge Management (EKNOW'09)* à Cancun, au Mexique, en février 2009. Cet article visait à introduire le modèle de flux de connaissances, soit la base théorique de cette thèse de doctorat. Cet article est d'ailleurs une version préliminaire de l'article de revue présenté au chapitre 4.

Le chapitre 3 présente le premier article de revue. Intitulé "A qualitative and quantitative data collection methodology for knowledge analysis in software engineering", et écrit par Olivier Gendreau et Pierre N. Robillard, il a été soumis pour publication dans un numéro spécial sur la recherche quantitative en génie logiciel de la revue *Empirical Software Engineering*. Cet article a pour principal objectif la présentation et la justification de la méthodologie utilisée dans le cadre des travaux de recherche de cette thèse. Plus précisément, l'article détaille la méthodologie ATS (*activity time slip*), une approche à partir de laquelle des développeurs logiciels doivent enregistrer leurs activités dans une perspective de connaissances. Les données recueillies sont

ensuite codifiées selon le modèle de flux de connaissances, qui est inspiré du modèle de création de connaissances de Nonaka & Takeuchi. Une étude de cas multiples est présentée, afin de démontrer l'originalité et la pertinence de la méthodologie proposée.

Le chapitre 4 présente le second article de revue. Intitulé "Consequences of Documentation Quality in FLOSS Reuse: A Case Study", et écrit par Olivier Gendreau et Pierre N. Robillard, il a été soumis pour publication à la revue *Information and Software Technology*. Cet article présente l'utilisation du modèle de flux de connaissances, dans le cadre d'une étude de cas, afin d'analyser les conséquences de la qualité de la documentation lors de la réutilisation de composants FLOSS (*free/libre open source software*). L'analyse de l'étude de cas permet de déterminer les conséquences négatives d'une documentation inadéquate sur le flux de connaissances au sein d'un projet de développement logiciel.

Le chapitre 5 présente le troisième et dernier article de revue. Intitulé "Is Design Useful in Small Software Projects? An Exploratory Case Study", et écrit par Olivier Gendreau et Pierre N. Robillard, il a été soumis pour publication à la revue *Journal of Systems and Software*. Cet article a pour but de comprendre les mécanismes menant aux divergences observées entre la conception et l'implémentation d'un projet de développement logiciel. L'utilisation de la méthodologie ATS et du modèle de flux de connaissances facilite l'analyse de l'étude de cas.

Finalement, le chapitre 6 présente des résultats de recherche complémentaires, pas encore soumis pour publication. L'objectif du chapitre est de caractériser trois projets intégrateurs de développement logiciel, à la manière d'une étude de cas multiples de type exploratoire reposant sur la méthodologie ATS et la modélisation par flux de connaissances. D'abord, les caractéristiques générales des projets sont présentées. Par la suite, les développeurs sont caractérisés par l'analyse de leur production de jetons (*time slip*). Puis, l'effort est caractérisé sous plusieurs perspectives: la répartition de l'effort global, la répartition et l'évolution du travail individuel et participatif, le séquencement cognitif, ainsi que la relation entre l'effort et le code source.

CHAPITRE 3

A QUALITATIVE AND QUANTITATIVE DATA COLLECTION METHODOLOGY FOR KNOWLEDGE ANALYSIS IN SOFTWARE ENGINEERING

3.1 Abstract

To better understand the complexity of software development, it could be useful to analyze software activities from a knowledge perspective. However, the nature of knowledge offers a methodological challenge, since knowledge is the result of various cognitive activities and mostly resides in a software developer's mind. This paper proposes the activity time slip (ATS) methodology, which is an approach in which software developers record their activities from a knowledge viewpoint. The ATS methodology is designed to support the grounded theory approach. We present a multiple-case study analysis from four industrial capstone projects conducted between 2006 and 2009. Data are codified based on the knowledge flow model, which is related to Nonaka and Takeuchi's knowledge creation model. The level of accuracy obtained with the ATS methodology is sufficient to explore various knowledge perspectives in software development. The methodological challenges presented by both the participants and the researchers are discussed.

3.2 Introduction

Software engineering is a knowledge-intensive activity (Henninger, 1997; Robillard, 1999; Xu, Rajlich, & Marcus, 2005; Bjornson &Dingsoyr, 2008; Ras & Rech 2008), and software artifacts constitute an accumulation of knowledge owned by organizational stakeholders (Baetjer, 1998). Software development requires programmers to gather and absorb large amounts of knowledge distributed over several domains, such as application and programming (Clayton, Rugaber, & Wills, 1998), and to encode that knowledge in the software (Xu et al., 2005).

In order to better understand the complexity of software development, Ko, DeLine, & Venolia (2007) suggest analyzing software activities from a knowledge perspective. However,

the nature of knowledge offers a methodological challenge. Since knowledge is the product of various cognitive activities and mostly resides in a software developer's mind, it would perhaps be better described by the developers. Therefore, we propose an approach where software developers record their activities from a knowledge viewpoint.

Qualitative data can come from three sources: interviews, observations (live or audio-video), and artifacts. Interviews are limited and time-consuming for both the researchers and the participants involved. Live observation is often less invasive for the participants, but observation time is limited. Audio-video may be less invasive still, but a formal protocol analysis from scripts could be very time-consuming. Artifacts are often produced by the participants as part of their usual tasks.

This paper's main objective is to detail the activity time slip (ATS) methodology, a qualitative and quantitative research data collection technique, which allows the analysis of software development from a knowledge perspective. The ATS methodology is inspired from the work diary (Lethbridge, Sim, & Singer, 2005), and provides a good tradeoff between data accuracy and data analysis effort. It makes it possible to capture complex information in a flexible way, on an ongoing basis, and in the developers' real environment.

The ATS methodology is designed to support the grounded theory approach, which supposes that theory is ''grounded'' in the data, rather than presumed at the outset of the research. In pure grounded theory, there would be no preconceptions with respect to the concepts of importance. Often researchers adapt grounded theory, by using prior software engineering knowledge based on expert opinion and the scientific literature as a starting point for domains and probes in the preliminary study proposal. Grounded theory is based on two major principles: first, that phenomena are not conceived as static, but rather constantly changing in response to evolving conditions; and second, that people have, although do not always use, the means to control their destinies by their response to conditions (Corbin, 1990).

Although it is not the purpose of this paper to provide an in-depth and comprehensive review of grounded theory, some of its basic tenets should be understood, as they provide the scientific rationale for an approach such as the ATS methodology.

These tenets are the following:

- Data collection and analysis are interrelated and concurrent processes, rather than linear ones; analysis begins as soon as the first bit of data is collected.
- Concepts are the basic units of analysis. Thus, data collected from subjects are given conceptual labels.
- Specificity of the concept is achieved by understanding the qualifiers of the concept (e.g. what factors impact the concept, such as the input artifact, the type of activity, interaction).
- Analysis is achieved through constant comparison of similarities and differences in the data, and the search for both supportive and disconfirming evidence. Throughout the research process, hypotheses are revised based on the ongoing assessment of both qualifying and disqualifying evidence derived from partial data analysis, until they can be fully supported by all the data, facilitating a robust analysis.
- Sufficient data must be collected to reach "conceptual saturation", the complete elaboration of the properties, dimensions, and variations that constitute each category or theme.

The ATS methodology is developed to gain an understanding of how knowledge needs evolve throughout a software project's development. This methodology is illustrated with a multiple-case study analysis from four industrial capstone projects conducted between 2006 and 2009 at the École Polytechnique de Montréal.

The structure of this paper is as follows. Section 2 presents the ATS methodology. Section 3 details knowledge concepts. Section 4 offers a knowledge flow analysis of software development. Section 5 provides a discussion of our research. Finally, our conclusions are presented in section 6.

3.3 Methodology

3.3.1 Case study

Software engineering involves real people in real environments (Lethbridge et al., 2005). Conducting empirical research on real-world issues implies a tradeoff between level of control

and degree of realism. Case studies are, by definition, conducted in real-world settings, and thus have a high degree of realism, mostly at the expense of the level of control, which makes them suitable candidates for a software engineering research methodology (Runeson & Höst, 2009). According to Yin (2003), a case study involves a *how* or *why* form of research question, does not require control of behavioral issues, and focuses on contemporary events.

Since we want to understand the flow of knowledge throughout a software project's development, we need to accept a lower level of control in order to better seize the realism of the process. Therefore, the case study is the appropriate research methodology for our purpose.

It is important to use several data sources in a case study, in order to limit the effects of one interpretation of a single data source (Runeson & Höst, 2009). A single study will have a large number of parameters, some controlled and some completely unconstrained (Miller, 2008). However, the evidence from multiple cases is often considered more compelling, and the overall study is therefore regarded as more robust (Herriot & Firestone, 1983). Moreover, multiple cases allow consideration of replication logic, which is analogous to that used in multiple experiments (Hersen & Barlow, 1976). Yin (2003) states that more than two cases can already make a strong argument. Therefore, we chose to analyze four different projects as part of a multiple-case study.

Robson (2002) distinguishes four research purposes: exploratory, descriptive, explanatory, and improving. Exploratory research aims to find out what is happening, seeking new insights and generating ideas and hypotheses for new research. Descriptive research is designed to portray a situation or phenomenon. Explanatory research seeks an explanation of a situation or a problem, mostly in the form of a causal relationship. Improving research tries to enhance some aspect of the phenomenon studied.

Case study methodology was originally used primarily for exploratory purposes, and some researchers still limit case studies for this purpose (Flyvbjerg, 2006). By trying to better understand the knowledge flow in software development, our research purpose is exploratory.

Other research methodologies related to case studies include the survey, the experiment, and action research (Runeson & Höst, 2009). A survey is a collection of standardized information from a specific population. Its primary objective is descriptive and its primary data are quantitative. An experiment is characterized by measuring the effects on one variable of manipulating another variable. Its primary objective is explanatory and its primary data are

quantitative. Action research aims to influence or change some aspect of the focus of the research (Robson, 2002), and is closely related to the case study. Its primary objective is improvement and its primary data are qualitative.

Data collected in an empirical study may be quantitative or qualitative. Quantitative data involve numbers, while qualitative data are represented as words and/or pictures (Gilgun, 1992). Qualitative research has been designed mostly by educational researchers and other social scientists to study the complexities of human behavior (Taylor & Bogdan, 1984). It could be argued that the study of human behavior is one of the few phenomena that is complex enough to require qualitative methodologies. In software engineering, the blend of technical and human behavioral aspects requires a combination of qualitative and quantitative methodologies, in order to take advantage of the strengths of both (Seaman, 1999).

The ATS methodology makes use of both qualitative and quantitative data. For instance, a description of software activities from a knowledge perspective (qualitative data) and the time spent on these activities (quantitative data) are important aspects of our methodology.

Theory generation methodologies are generally used to extract a statement or proposition from a set of field notes that is supported in multiple ways by the data (Seaman, 1999). These methodologies are often referred to as grounded-theory methodologies, because the theories, or propositions, are grounded in the data (Glaser & Strauss, 1967). To illustrate the potential of the ATS methodology for developing grounded theory, we have developed a knowledge flow model (cf. section 3.5).

3.3.2 Data collection techniques

Field studies provide empirical study researchers with a unique perspective on software engineering, and are particularly useful in understanding practices and in developing theories.

Lethbridge et al. (2005) provide a taxonomy of data collection for software engineering field studies based on the degree of human contact required. First degree contact techniques, such as interviewing and shadowing, require direct access to a participant population. Second degree contact techniques, such as fly-on-the-wall, allow researchers to observe work without needing to communicate directly with participants. Finally, third degree contact techniques, such as tool analysis, use logs, and documentation analysis, require access only to work artifacts.

Of course, close contact with subjects (first-degree techniques) requires a stronger working relationship than the unobtrusive study of work artifacts (third-degree techniques). However, first-degree techniques are invaluable because of their flexibility and the phenomena they can be used to study, which are mainly related to cognition. However, their two major drawbacks are that they are less reliable and consume more resources than third-degree techniques.

First-degree techniques can be either inquisitive or observational (Lethbridge et al., 2005). Each type is appropriate for gathering a different kind of information from software engineers. Inquisitive first degree techniques allow the experimenter to obtain a general understanding of the software engineering process, such as brainstorming, interviews, questionnaires, and conceptual modeling. Observational first degree techniques provide a real-time portrayal of the phenomena studied, such as work diaries, think-aloud protocols, shadowing, and participant observation. However, it is more difficult to analyze the data, both because they are dense and because they require considerable knowledge to interpret correctly.

The ATS methodology is based on a data collection technique which can be employed for the entire duration of projects (without sampling). We want to achieve the best tradeoff between data accuracy and data analysis effort. Observational first degree techniques are the most likely to meet this requirement. Indeed, Lethbridge et al. (2005) conclude that interviews and questionnaires (inquisitive techniques) are the most straightforward instruments, but the data they produce typically present an incomplete picture. Bonke (2005) compared diary information (observational technique) to questionnaire information, and found that time-use information is preferentially obtained from diaries, as this methodology is considered more reliable than information from questionnaires.

Several studies aimed at understanding software development work practices have employed different methodologies. Perry et al. (1994) used time diaries (13 developers in one year) to understand how technology affects software process. Singer, Lethbridge, Vinson, & Anquetil (1997) studied the work practices of software engineers by surveying, observing (14 half-hour sessions), and interviewing developers. LaToza, Venolia, & DeLine (2006) conducted two surveys and a semi-structured interview to understand code-related activities and the motivation behind those activities. Chong & Siino (2006) explored interruption patterns among software developers who program in pairs versus those who program solo by analyzing 40 hours

of direct observation data. Ko et al. (2007) also analyzed work interruptions by employing the shadowing technique with seventeen developers and transcribed their activities minute by minute in 90-minute sessions, for a total of 25 hours of work. The analysis of shadowing data is very time-consuming. But, the work diary technique presents a good tradeoff between data accuracy and data analysis effort.

3.3.2.1 Work diary

Work diaries require respondents to record events that occur during the day. This may involve filling out a form, recording specific activities as they occur or at the end of the day, or noting a current task occurring at a preselected time (Lethbridge et al., 2005).

The main advantage of time diary information is that very complex information is provided in a very flexible way (Bonke, 2005). Diary studies have high ecological value, as they are carried out *in situ* in the users' real environments (Czerwinski, 2004). Work diaries can provide better event self-reporting, because they record tasks on an ongoing basis rather than in retrospect. Moreover, this method gives researchers a way of understanding how software engineers spend their time without undertaking a great deal of observation or shadowing (Lethbridge et al., 2005). Perry et al. (1994) were able to validate the time diary as a low-cost, effective way to determine how people spend their time.

However, there are three major drawbacks associated with work diary entries. They rely on self-reporting, which may not always represent reality. They can interfere with the activity of respondents as they work. Participants may fail to record certain events, or may not record events with sufficient detail (Lethbridge et al., 2005).

The next section describes how these drawbacks are managed in the ATS methodology.

3.3.2.2 Activity Time Slip (ATS)

The ATS data collection technique is inspired from the traditional work diary. Analyses based on the preliminary ATS methodology were first published by Germain & Robillard (2005) and improved by Gendreau & Robillard (2007, 2009). The meaning of *work* (from work diary) is different from that of *activity* (from the ATS). On the one hand, work is related to a task, as usually defined by a project manager. Examples of tasks reported in a work diary could be:

coding module A, testing B, etc. It is often part of a schedule and is related to project resources. On the other hand, an activity is a personal endeavor undertaken while a developer is executing a task. Examples of an activity reported in an ATS are: browsing the Web, reading API-X, talking to John about ABC, etc. A task may include many activities, but an activity only relates to one task. An activity relates to the *real* effort invested in software development, while a task relates to what is *prescribed*.

The ATS methodology requires that, each time a developer executes an activity, details must be logged in an ATS token. Table 1 details the token fields.

Table 3.1: Activity Time Slip (ATS) Token Content

Field	Description
ID	Unique token identifier
Date	Activity date
Start time	Activity start time
End time	Activity end time
Effort	Activity duration (computed from the start/end time fields)
P_1 .. P_n	P_1 to P_n participants involved in executing the activity
It	Activity iteration identifier
Input artifact	Activity main input artifact
Output artifact	Activity main output artifact
Discipline	Process discipline related to the activity
Role	Process role of the developer who executed the activity
Process activity	Process activity related to the activity
Activity description	Detailed description of the activity

The ATS methodology is more reliable than the work diary approach, because it deals with its drawbacks. While work diaries mainly compute task duration (elapsed time often on days scale), the ATS method aims to record actual effort expended on activities (time duration on hours scale). Each developer uses a preformatted spreadsheet to detail activities on an ongoing basis, providing a precision of roughly one token per hour. This approach minimizes interference with work, and encourages participants to record every activity without affecting accuracy. Moreover, a member of the research team regularly validates tokens by making sure the developers fill out each field of the token completely. This was done in order to closely represent the real effort

expended on the various activities. Interviews were conducted with the participants to understand their difficulties in meeting the requirements of the ATS methodology, and short training sessions were provided when required. Furthermore, a consistency validation was also conducted on the meaning of the token.

3.3.3 Research data

Data for our research were collected from four industrial capstone projects conducted between 2006 and 2009 at the École Polytechnique de Montréal. The capstone project is an elective project-oriented course for senior software engineering students. The four projects were based on requirements supplied by a single avionics industrial partner. An engineer from the participating organization met with the students once a week to assist them in developing the software product. The collocated software development team had access to a private development room on campus for the duration of the project, equipped with a meeting table, a whiteboard, and five workstations.

The team was formed based on four criteria: current number of cumulated credits, internship experience in industry, current grade point average (GPA), and grades for software design and software process courses.

The capstone project was conducted by a team of 4 or 5 developers over one semester (14 weeks) on a fixed schedule of three half-day collocated sessions and a flexible schedule of up to three extra half-days per week.

3.3.4 Ethical considerations

According to Runeson & Höst (2009), research involving key ethical factors include: informed consent, review board approval, confidentiality, handling of sensitive results, inducements, and feedback.

Students participating in a capstone project signed a consent form explaining the ATS methodology and ensuring the confidentiality and proper handling of their personal data. The research methodology and the consent form were previously approved by the École Polytechnique de Montréal's ethics committee (certificate CÉR-05/06-15).

3.4 Knowledge Perspective

The subjects presented in this section show that knowledge is complex. It can be viewed from many perspectives and modeled accordingly. Knowledge management, which is largely interdisciplinary, is becoming a real concern in software engineering. New methodologies must be explored to enable us to better capture data related to knowledge flow.

3.4.1 Knowledge models

In the cognitive sciences, there are four knowledge models that are referred to widely: Kolb's model of experiential learning, Argyris and Schön's double-loop learning theory, Wenger's theory of communities of practice, and Nonaka and Takeuchi's theory of knowledge creation (Bjornson & Dingsoyr, 2008).

Kolb (1984) describes learning from experience as involving four learning modes that can be placed in two dimensions. One dimension relates to how people grasp experience and includes two modes: comprehension (abstract conceptualization) and apprehension (concrete experience). The other dimension relates to how people convert experience, and also includes two modes: intention (reflective observation) and extension (active experimentation). Kolb stresses the importance of taking advantage of all four modes for learning to be effective.

Argyris & Schön (1978) differentiate single- and double-loop learning. Single-loop learning involves the generation of new action strategies to achieve existing governing values, while double-loop learning involves adaptation and modification of the governing values themselves (Dick & Dalmau, 1990).

Wenger (1998) opposes the traditional individual learning process in favor of a social type of learning process, which he refers to as communities of practice (CoP). The members of a CoP engage in a process of collective learning through the action of sharing knowledge related to a common practice.

Nonaka & Takeuchi (1995) claim that knowledge creation occurs through the interaction between tacit and explicit knowledge, which is called knowledge conversion. Polanyi (1967) categorized knowledge as either explicit or tacit. Explicit knowledge can be expressed in formal and systematic language, and can be processed and stored relatively easily (Williams, 2006). Tacit

knowledge is deeply rooted in an individual's actions, experience, and values, making it highly personal and difficult to formalize (Schon, 1983). There are four types of knowledge conversion: socialization (from tacit knowledge to tacit knowledge), externalization (from tacit knowledge to explicit knowledge), combination (from explicit knowledge to explicit knowledge), and internalization (from explicit knowledge to tacit knowledge). Socialization relates to the conversion of new tacit knowledge from past experiences. Externalization is the process of crystallizing knowledge by making tacit knowledge explicit. Combination relates to converting explicit knowledge to more complex or systematic explicit knowledge. Internalization occurs when someone embodies explicit knowledge in tacit knowledge.

3.4.2 Knowledge management

Knowledge management is a large interdisciplinary field (Bjornson & Dingsoyr, 2008). Earl (2001) proposes taxonomy of strategies, or "schools", categorized as: technocratic, economic, and behavioral. The technocratic schools are: the systems school, focusing on technology for knowledge-sharing; the cartographic school, which is concerned with mapping organizational knowledge; and the engineering school, which focuses on processes and knowledge flows in organizations. The economic school is concerned with commercial exploitation of knowledge and intellectual capital. The behavioral schools are: the organizational school, focusing on networks for sharing knowledge; the spatial school, which focuses on how office space can be designed to promote knowledge-sharing; and the strategic school, which sees knowledge management as a dimension of competitive strategy.

The engineering school, focusing mainly on process, is the knowledge management school receiving the most empirical attention (Bjornson & Dingsoyr, 2008). Two major categories can be identified within this school. The first investigates the entire software process with respect to knowledge management. The second considers possibilities of specific activity improvement to the software process.

As part of the first category, Alavi & Leidner (2001) believe the major challenge in knowledge management is to facilitate the flow of knowledge between individuals in order to maximize the amount of knowledge transfer.

Arent & Norbjerg (2000) studied software process improvement (SPI) from a knowledge perspective based on the knowledge creation model (Nonaka & Takeuchi, 1995). They concluded that both tacit and explicit knowledge are crucial to SPI success. Tacit knowledge is necessary to change practices, and explicit knowledge is necessary to create an organizational memory.

Nerur & Balijepally (2007) argue that the type of software process affects how knowledge is managed. The traditional (engineering-based) approach relies primarily on managing explicit knowledge, while agile methodologies primarily rely on managing tacit knowledge.

Dahkli & Chouikha (2009) propose a knowledge-oriented software development process designed to reduce the knowledge gap resulting from the discrepancy between the knowledge integrated in software systems and the knowledge owned by organizational actors.

As part of the second category, Melnik & Maurer (2004) discuss the role of conversation and social interaction as the key elements of effective knowledge-sharing in an agile process. They conclude that explicit knowledge-sharing is inefficient when complex cognitive artifacts are used. The higher the level of complexity, the more need there is for interactive knowledge-sharing through direct verbal communication.

Bjornson & Dingsoyr (2005) investigated knowledge-sharing through mentoring in a small software consultancy company. In order to improve mentoring, they propose introducing methods to increase the employees' level of reflection.

Desouza, Awazu, & Wan (2006) examined what factors contribute to the use of explicit knowledge in a software engineering organization. They found that perceived complexity, perceived relative advantage, and perceived risk are factors affecting the use of explicit knowledge.

To sum up, knowledge management's engineering school literature investigates software process and practices with respect to knowledge management, but not from a knowledge flow perspective.

3.5 Knowledge Flow in Software Projects

The following section illustrates the uses of the ATS methodology to derive a conceptual model based on the grounded theory, as well as to validate the model with multiple-case studies.

Based on the grounded theory, the ATS tokens of the 2006 capstone project provided the concepts on which we can develop a knowledge flow model (Figure 3.1), which is related to Nonaka and Takeuchi's knowledge creation model. We then used this knowledge flow model to codify each token of the four capstone projects, in order to analyze software development from a knowledge perspective.

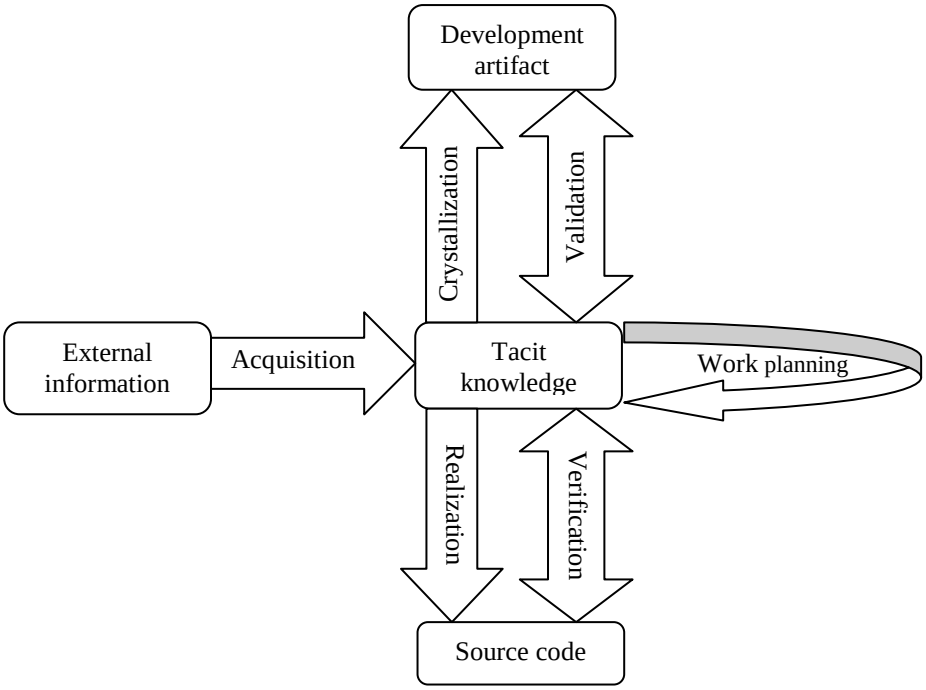

Figure 3.1: Knowledge Flow Model

The four rounded-corner boxes in Figure 3.1 represent knowledge sources. External information can be general or specific to the project under development. General information may come from various sources, such as the Web, a paper, or a book. Specific information comes from any of the project's pre-existing documentation. A development artifact is a physical representation of knowledge, such as a software requirement specification (SRS), a design specification, or a test plan. Source code strictly includes executable statements and comments. Tacit knowledge is individual knowledge built from interacting with other knowledge sources.

The arrows in Figure 3.1 represent the cognitive factors, which constitute the knowledge flow between knowledge sources. Real-life examples (based on the 2006 project) of all six cognitive factors are detailed in Table 3.2. The acquisition cognitive factor is involved when a developer needs to increase his tacit knowledge from external information. The crystallization cognitive

factor is the translation of a developer's mental representation of a concept (tacit knowledge) into an artifact (explicit knowledge), such as a use-case diagram or an architectural plan. The realization cognitive factor also involves the translation of tacit knowledge into explicit knowledge, but requires, in addition, technical know-how, which is related to source code production. The validation cognitive factor involves bidirectional knowledge flow between tacit knowledge and development artifacts (explicit knowledge), in order to validate the consistency of those two knowledge sources. The verification cognitive factor is like validation, except that source code is the knowledge source, thus involving technical know-how. The work planning cognitive factor mostly involves developers' synchronization of the project's planning and progress knowledge.

This knowledge flow model is limited to software development activities. The management activities related to the software project are not taken into consideration in this model, because they are not specific to software development and they frequently involve several projects. Writing a software development plan is an example of a management activity.

Table 3.2: Token examples

Cognitive factor	Token Description
Acquisition	Read Qt website documentation to better understand drag-and-drop functions.
Crystallization	Define use cases 1 and 2 in the Use Case Specification.
Realization	Code CGraphicComponents and CGraphicDesign classes for drag-and-drop functionalities.
Verification	Fix drag-and-drop bug in CGraphicComponents and CGraphicDesign classes.
Validation	Conduct team peer review of the architecture document.
Work planning	Conduct team meeting for iteration 3 task planning.

In order to extract knowledge behavior from ATS tokens, a coding scheme, based on the knowledge flow model, has been designed. An ATS token is codified according to the cognitive factor concerned. However, some tokens involve more than one cognitive factor. Therefore, the coder needs to determine the dominant cognitive factor, mainly based on the description of the token and its context (input artifact, role, process, etc.). For instance, fixing a code defect involves both the verification and realization cognitive factors. First, it requires locating the defect in the code, which is related to the verification cognitive factor. Then, the actual fixing of

the code involves the realization cognitive factor. In this situation, the dominant cognitive factor remains verification.

All the tokens of the four capstone projects were codified by two independent coders, who had to decide which cognitive factor was dominant. However, tokens related to academic and technical activities were not accounted for in the codification, since they were not specific to project development. Academic activities are related to the academic course, such as teamwork training and project presentation. Technical activities are related to tasks which can be performed by technicians, such as configuring the network or setting up and maintaining the development environment.

From a coding scheme viewpoint, there are 6 possible software development (SD) categories: acquisition, crystallization, verification, validation, realization, and work planning. There are also 3 possible categories not related to software development: management, academic, and technical. Table 3.3 presents the software development (SD) effort, the number of SD tokens, and the token-per-hour ratio.

Table 3.3: Software development effort and tokens

Project	SD effort (h)	SD tokens	Token/hour
P06	997	1426	1.4
P07	750	1408	1.9
P08	810	887	1.1
P09	628	621	1.0

The codification of ATS tokens allows the analysis of effort distribution, which makes it easier to understand a project's knowledge flow.

Every project studied had two deadlines. The first required the development team to present their system architecture to the industrial partner. It occurred at between 30% and 45% of project completion, depending on the project. The second deadline occurred at the end of the semester, when the product was delivered to the client.

Figures 3.2 to 3.5 present the total effort expended on each cognitive factor in relation to project completion for projects P06, P07, P08, and P09. Each of the 6 curves of the graphs represents the

total effort expended (Y-axis) for a given cognitive factor with respect to the percentage of project completion (X-axis). For example, in P06 (Figure 2), at 20% of project completion (X-axis), 11% of the total effort (Y-axis) had been expended on crystallization. Validation represented 5% of the total effort and acquisition 3%, realization and work planning both accounted for 1%, and no verification effort had been expended to that point. The analysis of the slopes of the 6 curves in Figure 2 provides a better understanding of the relationship between cognitive factors throughout the project.

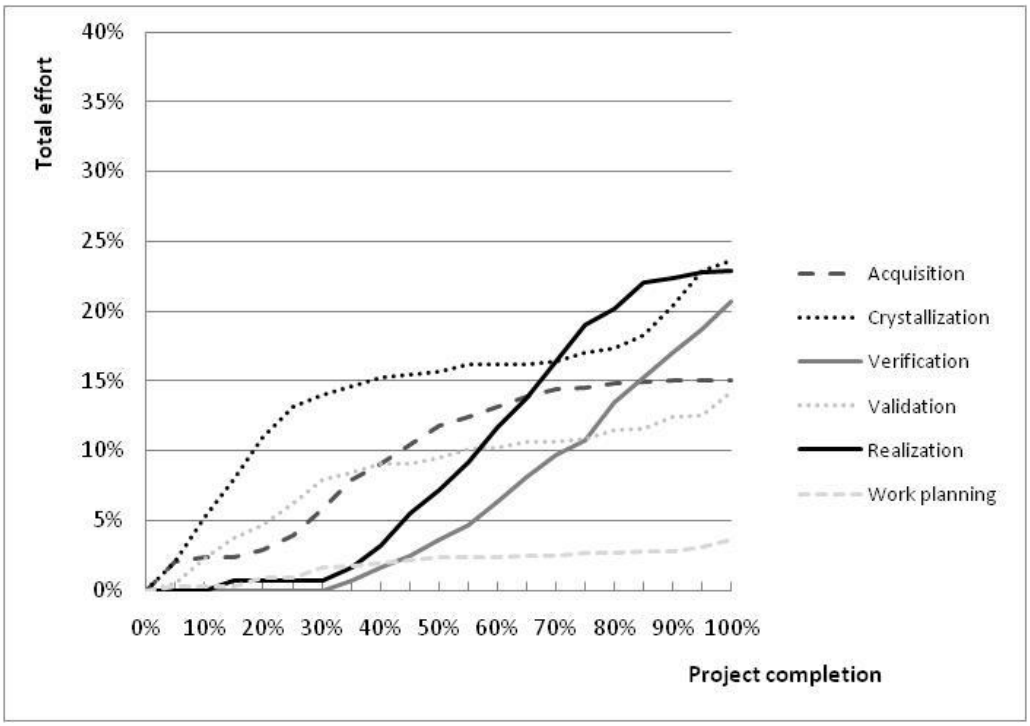

Figure 3.2: Project P06 total effort distribution

As depicted in Figure 3.2, at the end of the first phase of project P06 (30% of project completion), crystallization was the project's most important cognitive factor (14% of total effort), requiring an almost constant effort, as can be observed from the linear part of the cumulative effort curve. This behavior is typical of disciplined software processes, because many development artifacts are produced during that timeframe, such as SRS, use case documentation, and architecture and design documentation. Between 85% and 95% of project completion, crystallization is an important cognitive factor. This behavior can be explained by the developers'

need to update artifacts to fit the software product before delivery, which we refer to as *retrofitting*.

Throughout project P06, validation is closely related to crystallization, which is typical of software projects, since validation mostly depends on reviewing crystallized development artifacts.

Acquisition is an important cognitive factor at between 25% and 65% of project completion. It is related to new knowledge needed from developers in order to understand the project's domain or elaborate the system architecture.

Work planning is not a significant part of the project's total effort, but it requires an almost constant effort for the duration of the project, as can be seen from the cumulative effort curve throughout the project, which is almost linear.

Realization and verification are closely related throughout the project. Both start by being important cognitive factors at around 35% of project completion, which is close to the end of the first phase of the project. Realization is the most important cognitive factor from 35% to 85% of completion, while verification is very important from 35% until the end of the project. The relation between the two cognitive factors is due to code verification and testing being complements to code implementation.

Figure 3.3 illustrates the cognitive factor effort distribution for project P07. The effort distribution for the first phase of this project (35% of project completion) is very similar to that of P06: crystallization was the project's most important cognitive factor (13% of total effort), validation is closely related to crystallization, acquisition is moderately important (5% of total effort), and realization and verification are unimportant (2% and 1% of total effort respectively). However, by the end of the project, P07 and P06 were noticeably different. Work planning added up to 11% of total effort, and crystallization added up to 26% of total effort, a particularly high figure. Moreover, even though realization and verification were closely related throughout the project, verification surpassed realization at 90% of project completion, which is atypical. These observations can be explained by the team's development philosophy. Artifact production and work planning were considered as goals to achieve at "any cost". Consequently, the essence of software development – to develop a quality product – was overshadowed.

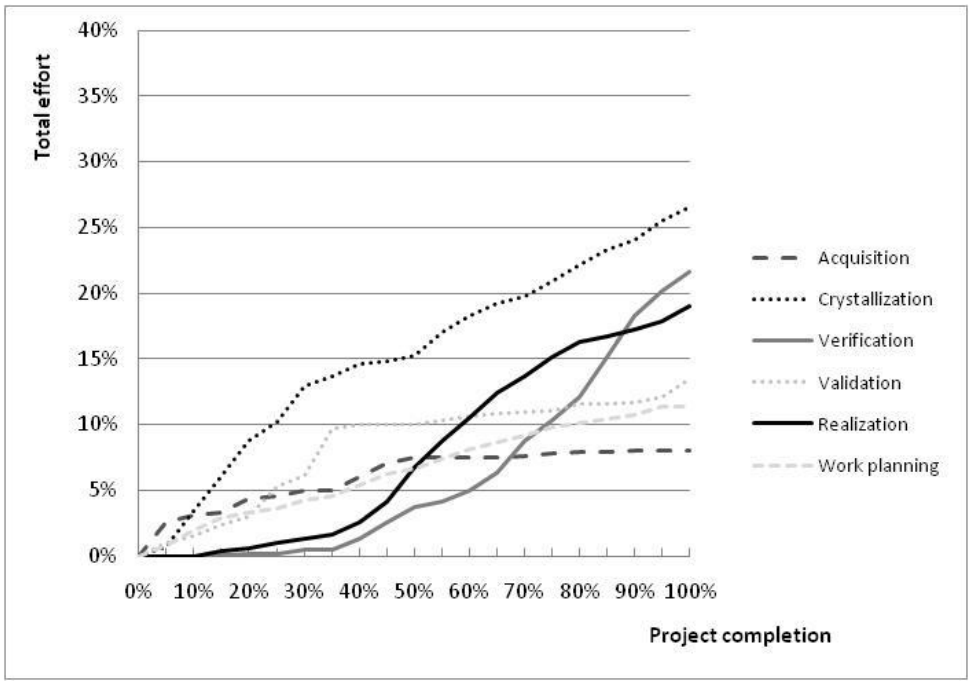

Figure 3.3: Project P07 total effort distribution

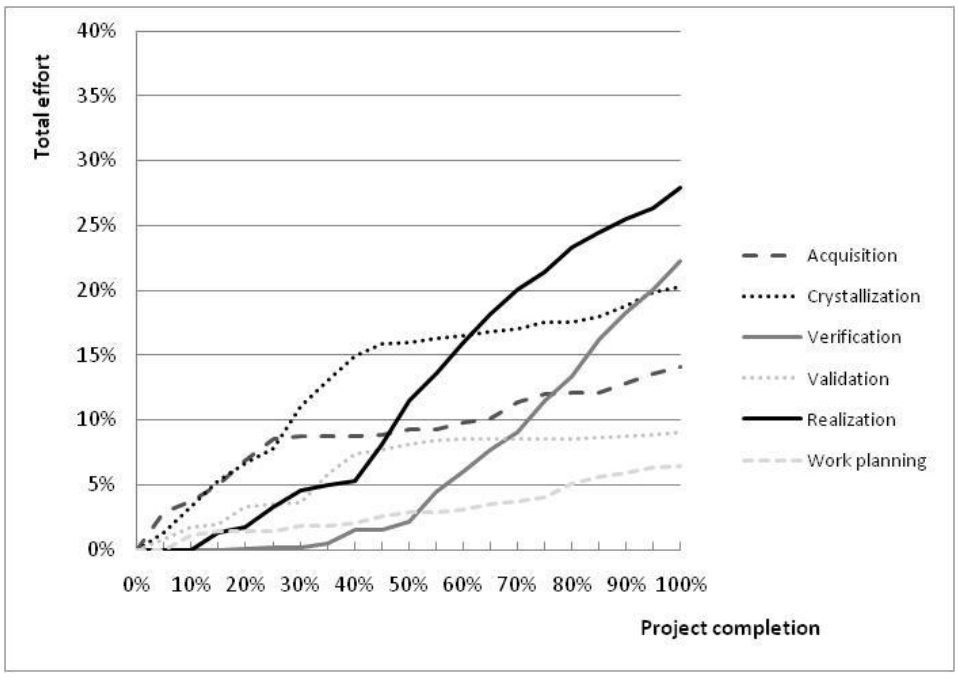

Figure 3.4: Project P08 total effort distribution

Figure 3.4 shows cognitive factor effort distribution for project P08. As was the case for projects P06 and P07, at the end of the first phase of P08 (45% of project completion), crystallization was the project's most important cognitive factor (16% of total effort). Like P06 and P07, throughout

the P08, validation was closely related to crystallization, and realization was closely related to verification. Work planning also required an almost constant effort for the duration of the project. However, P08's acquisition (9% of total effort of the project's first 25%) was completed sooner than in P06 and P07. This behavior can be explained by the developers' lack of domain-specific knowledge.

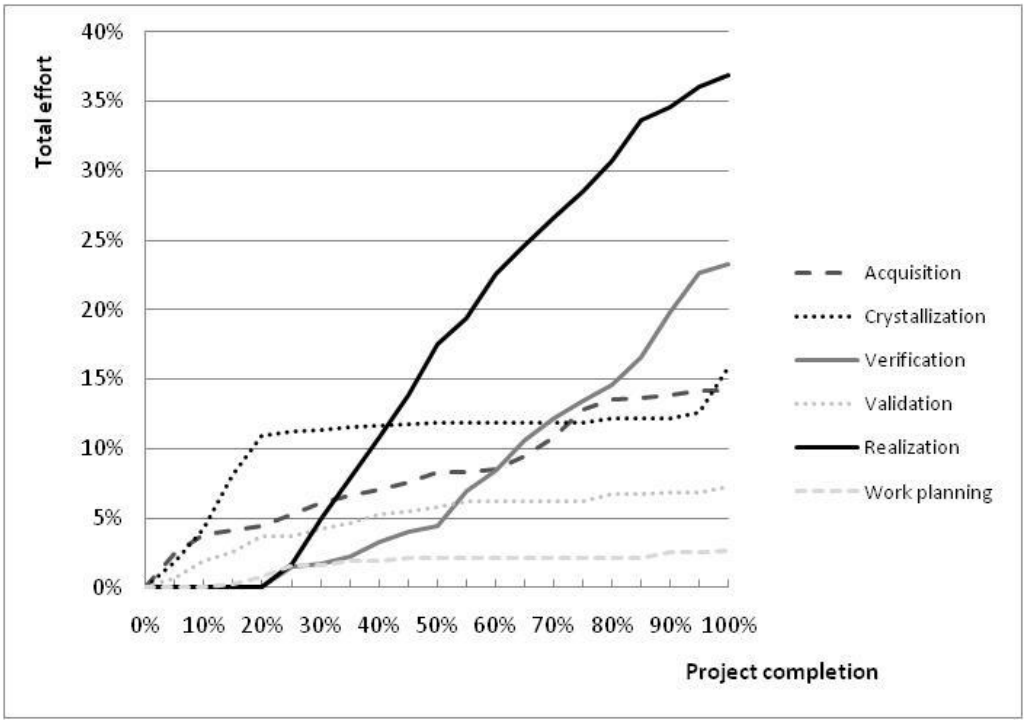

Figure 3.5: Project P09 total effort distribution

Figure 3.5 details cognitive factor effort distribution for project P09. As was the case for P06, P07, and P08, at the end of the first phase of P09 (35% of project completion), crystallization was the project's most important cognitive factor (12% of total effort). However, unlike the other projects, realization was already important (8% of total effort), and a crystallization plateau was reached at around 20% of project completion. Like the other projects, throughout P09, validation was closely related to crystallization, and realization was closely related to verification. But P09's realization (37% of total effort) is very important compared to that of P06, P07, and P08 (respectively 23%, 19%, and 28% of total effort). These observations can be explained by the team's development philosophy, which was very different from that in P07. The P09 developers rightly produced artifacts and planned their work in order to help software development. In other

words, the development of a quality product was the main focus of the project, and this was complemented by supporting activities, which was not the case for P07.

3.6 Discussion

In this section, we discuss the methodological challenges and results of the knowledge flow analysis.

3.6.1 Methodological challenges

The ATS methodology offers participant (project developer), researcher, intercoder reliability and external validity challenges.

3.6.1.1 Participant

The participant must understand what an activity is in order to accurately log its details in a token (cf. Table 3.1). A short training session provided to participants prior to the project by the researchers is usually enough. One difficulty for the participant is to identify an activity switch and log it accordingly. For example, the implementation of a feature mostly involves coding and testing/debugging activities that are intertwined in time. The participant must understand what he is doing and correctly log these activities.

3.6.1.2 Researcher

The researcher must consider three methodological challenges: continuous project methodological supervision, consistency validation, and an adequate coding scheme.

The researcher must provide the participant with continuous methodological supervision throughout the project. He has to regularly ensure that the tokens were correctly filled in by participants. The most common problems are an incomplete activity description and an inappropriate token duration. The first occurs mostly at the beginning of a project, and is easily fixed by appropriate participant training. The second is influenced by a participant's personality, as participants with a rigorous mind pay greater attention to activity switching than others. Close supervision is therefore required to motivate those participants to correctly fill in tokens.

Consistency validation is another critical aspect to consider. The researcher must validate tokens to ensure that they do not overlap. Two types of validation are performed: start/end time and participants. The start time and end time fields of each token are analyzed to make sure that they do not overlap. Also, fields P_1 to P_n (participants involved in the execution of an activity) are validated to make sure that every participant mentioned in a token did, in fact, complete a token confirming the same activity. For example, if participant A completes a token stating that he was in a meeting with participant B, and then a token from participant B must state that he was in a meeting with participant A.

The definition of the coding scheme is a methodologically critical aspect of any methodology based on protocol analysis. Our coding scheme is based on the cognitive factors of the knowledge flow model, which was developed using grounded theory. However, some tokens involve more than one cognitive factor. Therefore, the coder has to determine the dominant cognitive factor based on the token's description and context. To ensure the reliability of coding decisions, two independent coders were asked to codify the tokens of the four capstone projects. Thereafter, an intercoder reliability measure allowed assessment of the convergence of their coding.

3.6.1.3 Intercoder reliability

With judgment-based coding schemes, the best approach for validating the reliability of the quality of data is to rely on intercoder agreements (Perreault & Leigh, 1989).

When researchers use multiple coders and evaluate the convergence of their coding, the most commonly used measure of intercoder reliability is the simple percentage of agreement between two (or more) coders (Bettman & Park, 1980). Although easy to use, simple percentage agreement statistics are likely to be influenced heavily by the number of coding categories (Cohen, 1960).

The next most frequently used intercoder reliability measure is Cohen's kappa. However, this is a conservative measure because of the way it estimates interceders' agreement related to chance (Perreault & Leigh, 1989). It is difficult to operate in practice because it requires an estimate of the number of chance codings (Clark, 1999). The Perreault & Leigh reliability index (I_r) is

preferable to other methodologies, since it accounts for differences in reliability as a function of the number of categories (Kolbe & Burnett, 1991).

Table 3.5: Project reliability index

Project	I_r
P06	0.95
P07	0.93
P08	0.94
P09	0.93

Table 3.5 presents Perreault & Leigh's reliability index for the four projects based on two independent coders. The indices presented (all above 0.9) show a strong project coding reliability.

3.6.1.4 External validity

The external validity of empirical studies with students is a commonly raised concern. According to Carver, Jaccheri, Morasca, & Shull (2003), more and more students are employed during the summer or for complete internships in industrial environments, which brings an expanded set of skills to many upper-level courses. Höst, Regnell, & Wohlin (2000) conclude that only minor differences exist between students and professionals regarding their ability to perform relatively simple tasks requiring judgment. Moreover, their studies do not contradict the assumption that final-year software engineering students are qualified to participate in empirical software engineering research. Similar results were obtained in a study about detection methodologies for software requirement inspection conducted by Porter, Votta, & Basili (1995) with students and then replicated with professionals (Porter & Votta, 1998). Consequently, the external validity of our study is increased by relying on senior students who have some internship experience in the industry.

3.6.2 Knowledge flow results

As the ATS methodology enables representation of a model of knowledge flow throughout a software development project, the cognitive activity diagrams presented here serve to illustrate proofs of concept for the methodology.

The ATS methodology is a software activity measure which is independent of the software process used. It is related to recorded cognitive activities, which allow a better understanding of knowledge needs throughout a software project. A detailed analysis of these results is outside the scope of this paper.

3.7 Conclusions

There is a growing need to consider the knowledge perspective in software development, since developers' activities are mostly cognitive. Knowledge cannot be measured directly, since it is mostly tacit in the developer's mind. Techniques have been developed in the social sciences, like think-aloud protocols, to explore the participant thinking process. Such techniques can only be applied for a short period of time on very specific cognitive activities, however, and they are not well suited for studying the knowledge flow in a whole software development project.

The ATS methodology presented in this paper is a compromise between short duration, very accurate data (think-aloud) and project span self-reported data on participant cognitive activities. The level of accuracy obtained with the ATS methodology is sufficient to explore various knowledge perspectives in software development.

The ATS methodology offers challenges to researchers: appropriate project methodological supervision, consistency validation, and an adequate coding scheme.

This methodology is actually most useful when combined with other measurement approaches. Future work is needed in various directions. Tools can be built to ease the capture of data and to perform some of the validation online. ATS tokens can be combined with audio-video recording to increase the reliability and the content of the measure. ATS tokens can also be combined with records of technical activities performed on a computer.

CHAPITRE 4

CONSEQUENCES OF DOCUMENTATION QUALITY IN FLOSS REUSE: A CASE STUDY

4.1 Abstract

Context: Many software development projects reuse free/libre open source software (FLOSS) components for various reasons. However, FLOSS projects evolve quickly and the documentation does not always keep up.

Objective: This paper aims to assess the consequences of documentation quality for FLOSS component reuse.

Method: Data were gathered from an industrial capstone project, carried out over a 14-week period by five senior students, which required a total effort of roughly 1800 hours. The project's development was analyzed from a knowledge flow perspective, based on a professional constructivism approach, which integrates Nonaka and Takeuchi's knowledge creation model and Wertsch's social constructivist learning theory. The data gathering technique used was the effort time slip method.

Results: The main issues encountered in the industrial project were related to the ambiguous documentation of the reused FLOSS components. The industrial project helped to confirm that, in free/libre open source applications, effort is expended in software development, but not so much in documentation.

Conclusion: We suggest that the enforcement of a reusable code validation practice would lower FLOSS component reuse effort.

4.2 Introduction

Many software development projects reuse free/libre open source software (FLOSS) components for various reasons. Integrating software components which have been developed separately entails more than gathering such components from the marketplace and combining them (Merilinna & Matinlassi, 2006). It requires selection and evaluation of potential component

candidates. Component documentation has become a key issue in component reuse, because it is often the only way to assess the applicability, credibility, and quality of a third-party component (Taulavuori, 2002). However, the lack of documentation is one of the five fundamental problems associated with the current FLOSS development trend (Levesque, 2004).

FLOSS projects evolve differently from closed-source systems (CSS) (Koch, 2005). The total number of lines of code in FLOSS projects does not grow more rapidly than in CSS projects, but functions are added and modified more often in FLOSS projects over time (Paulson, Succi, & Eberlein, 2004).. This means that FLOSS projects evolve quickly and the documentation, when decent documentation is available, does not always keep up.

This paper aims to assess the consequences of documentation quality for FLOSS reuse components. Section 2 details the methodological approach, the knowledge flow perspective, and the effort time slip method. Section 3 presents the capstone project case study. Section 4 analyzes FLOSS documentation issues. Finally, section 5 extends our analysis by discussing the distribution of extra effort and recommending a software practice designed to minimize the consequences of inadequate documentation.

4.3 Methodological Approach

4.3.1 Capstone Project

Data for this study were gathered from an industrial capstone project carried out at the École Polytechnique de Montréal. A capstone project is a project-oriented course for senior software engineering students based on requirements supplied by an industrial partner. In this case, an engineer from the participating organization met with the students once a week to assist them in developing the software product. The collocated software development team had access to a dedicated room on campus for the duration of the project, furnished with a conference table, a whiteboard, and five workstations.

The team was formed based on four criteria: current number of accumulated credits, past internship experience in industry, and current grade point average, as well as software design and process course grades.

The capstone project was conducted over one semester (14 weeks) on a fixed schedule of three half-day team working sessions per week. The team workload was initially estimated to be 1800 hours.

The external validity of empirical studies performed by students is a commonly raised concern. According to Carver et al., more and more students are employed during the summer or in full internships in industrial environments, which brings an expanded set of skills to many upper-level courses (Carver et al., 2003). Höst et al. (2000) conclude that only minor differences exist between students and professionals regarding their ability to perform relatively simple tasks requiring judgment. Moreover, their study supports the assumption that final-year software engineering students are qualified to participate in empirical software engineering research. Also, a study on detection methods for software requirements inspection conducted by Porter et al. (1995) with students and then replicated with professionals produced similar results (Porter & Votta, 1998). Consequently, the external validity of our study is increased by relying on senior students who have some internship experience in the industry.

Precautions were taken to meet ethical requirements for research involving humans, particularly regarding informed consent and confidentiality. In this regard, an ethics certificate was issued for this research by the École Polytechnique de Montréal's ethics committee.

4.3.2 Knowledge Flow Perspective

The project's development was analyzed from a knowledge flow perspective, based on a professional constructivism approach integrating Nonaka & Takeuchi's (1995) knowledge creation model and Wertsch's (1985) social constructivist learning theory.

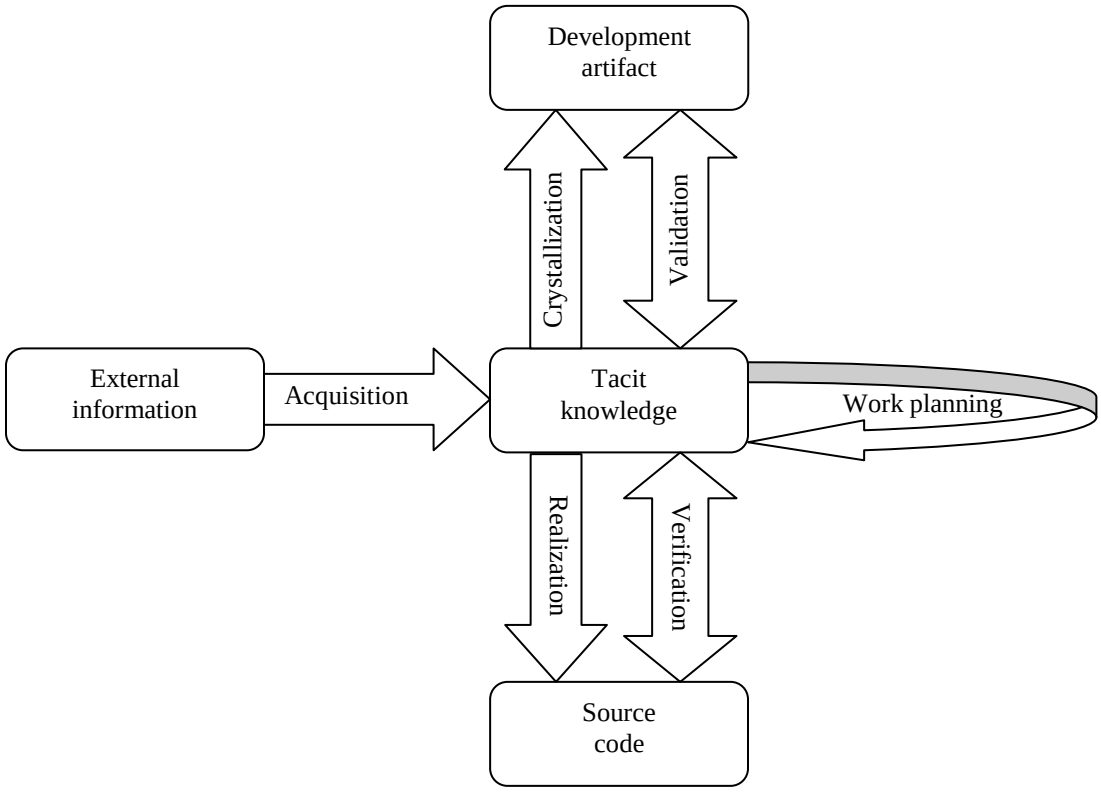

Figure 4.1: Knowledge Flow Model

The four rounded corner boxes in Figure 4.1 represent knowledge sources. External information can be either general or specific to the project under development. General information may come from various sources, such as the Web, a paper, or a book. Specific information can come from any preexisting documentation on the project. Development artifacts are physical representations of knowledge, such as software requirements specifications (SRS), design specifications, or test plans. Source codes are executable statements and comments. Tacit knowledge is individual knowledge constructed through interaction with other knowledge sources.

The arrows in Figure 4.1 represent the cognitive factors, which are the knowledge flow between knowledge sources. The acquisition cognitive factor is involved when developers need to increase their tacit knowledge from external information. The crystallization cognitive factor is the translation of developers' mental representation of a concept (tacit knowledge) into an artifact (explicit knowledge), such as a use-case diagram or an architectural plan. The realization cognitive factor also involves the translation of tacit knowledge into explicit knowledge, but

requires, in addition, technical know-how, which is related to programming skills. The validation cognitive factor involves bidirectional knowledge flow between tacit knowledge and development artifacts (explicit knowledge), in order to validate the consistency of those two knowledge sources. The verification cognitive factor is like validation, except that source code is the knowledge source, thus involving technical know-how. The work organization cognitive factor mostly implies developers' synchronization of project planning and progress knowledge.

This knowledge flow model is limited to software development activities. The management activities related to software projects are not taken into consideration in this model, as management activities are not specific to software development and frequently involve several projects.

4.3.3 Effort Time Slip Method

The data gathering technique used is the effort time slip method, which is a more detailed version of the work diary (Perry et al., 1994). The effort time slip method was proposed by Germain & Robillard (2005) and improved by Gendreau & Robillard (2007, 2009). Each time a developer executes a task, details must be logged in a time slip token. Table 4.1 details the token fields.

Table 4.1: Time Slip Token Content

Field	Description
ID	Unique token identifier
Date	Task date
Start time	Task start time
End time	Task end time
Duration	Task duration (computed from the start/end time fields)
P_1 .. P_n	P_1 to P_n participants involved in the execution of the task
It	Task iteration identifier
Input artifact	Task main input artifact
Output artifact	Task main output artifact
Discipline	Process discipline related to the task
Role	Process role of the developer who executed the task
Activity	Process activity related to the task
Task description	Detailed description of the task

In a state-of-the-art review on data collection techniques for software field studies, Lethbridge et al. (2005) present the advantages and disadvantages of the work diary approach. Work diaries can provide better self-reports of events, because they record activities on an ongoing basis rather than in retrospect. Moreover, they give researchers a way of understanding how software engineers spend their time without undertaking a lengthy observation process or shadowing. However, there are three major drawbacks associated with work diary entries. They rely on self-reporting, which may not always represent reality, and they can interfere with respondents as they work. Participants may fail to record some events, or they may record them with insufficient detail.

The effort time slip method is more reliable than the work diary approach, because it deals with its own drawbacks. While work diaries are primarily used for accounting purposes, recording assigned task durations (elapsed time), the effort time slip method aims to compute actual effort expended on a professional activity with maximal precision. A task is assigned by the project manager, design package ABC, for example, and is part of the project planning work breakdown structure. Assigned task durations are measured in days. Activities are performed by developers as part of their task, look on the Web for XYZ information, design component A classes, or code method B, for example. Activities are measured in hours or parts of hours, each developer using a preformatted spreadsheet to detail activities on an ongoing basis, providing a precision of roughly one token per hour. This approach minimizes interference with work and encourages participants to record every activity without affecting accuracy. Moreover, a member of the research team regularly validates token contents for coherence and accuracy, and a consistency validation is also conducted after the project has been completed.

4.3.4 Independent Data Codification

In order to facilitate effort time slip analysis, a coding scheme, based on the knowledge flow model, is designed. A time slip token is codified according to the cognitive factor concerned. However, some tokens involve more than one cognitive factor. In such a case, the dominant cognitive factor is determined based on the description of the token and its context (task, process, etc.). For instance, fixing a code defect implies both the verification and realization cognitive factors. First, the defect in the code, which is related to the verification cognitive factor, must be

found. Then, the realization cognitive factor is used to actually fix the code. In this situation, verification is usually the dominant cognitive factor.

With judgment-based coding schemes, the best approaches for improving the quality of data rely on evaluating the judgments of two (or more) independent coders (Perreault & Leigh, 1989).

When researchers use multiple coders and evaluate the convergence of their coding, the most commonly used measure of intercoder reliability is the simple percentage of agreement between two (or more) coders (Bettman & Park, 1980). Although easy to use, simple percentage agreement statistics are likely to be influenced heavily by the number of coding categories (Cohen, 1960).

Other than the simple percentage of agreement, the most frequently used intercoder reliability measure is Cohen's kappa. However, this is a conservative measure, because of the way it estimates intercoder agreement related to chance (Perreault & Leigh, 1989). It is difficult to operate in practice, because it requires an estimate of the number of chance codings (Clark, 1999). The Perreault & Leigh reliability index is preferable to other methods, since it accounts for differences in reliability as a function of the number of categories (Kolbe & Burnett, 1991).

The Perreault & Leigh reliability index will be discussed further in section 4.5.1.

4.4 FLOSS Component Reuse: The Case of the SFLphone Capstone Project

4.4.1 Project Context

The industrial partner for the capstone project was Savoir-Faire Linux (SFL), a company created to assist organizations wishing to take advantage of Linux's potential for their information systems. They specialize in Linux-based applications, such as Oracle, MySQL, Apache, Samba, Iproute2, Squid, OpenVPN, and Asterisk.

The main goal of the SFL project is to add videoconferencing capabilities to the SFLphone, which is an SIP/IAX2-compatible softphone for Linux. The SFLphone project's main goal is to create a robust enterprise class desktop phone. While it can serve home users as well, it has been designed to manage hundreds of calls per day. The SFLphone was released under Version 3 of the GNU General Public License. It is being developed by the global community and is maintained by SFL.

For the capstone project, SFL requested the addition of 33 functional requirements and 13 non functional requirements to the existing SFLphone application. In order to help the development team with prioritization, the functional requirements were classified as essential (24), desirable (8), or optional (1).

The requested requirements can be summarized as 6 features: two-way video conversation; three-way audio and video conferencing; audio and video flow synchronization; encoding and decoding of incoming and outgoing flows; multiple-flow mixing; compliance with H.263, SIP/SDP, RTP, IAX standards.

The project was developed in C/C++ on the Linux platform. Many open source tools were used, such as GIT for code version management, Eclipse as a development environment, and TRAC as a project managing and bug/issue tracking system. Skype, xChat, and emails were also used to communicate.

4.4.2 Disciplined Software Process

The development team followed a disciplined software process based on the Unified Process for EDUcation (UPEDU) (Robillard et al., 2003). Figure 4.2 depicts a generic process practice. A role is responsible for the outcomes of an activity. An activity needs at least one artifact as an input and will generate an artifact as an output.

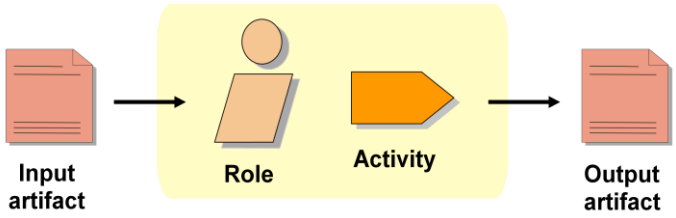

Figure 4.2: Generic Process Practice

Table 4.2 details the 7 disciplines, 20 activities, 8 roles, 12 main input artifacts, and 15 output artifacts of the process followed by the capstone project development teams.

Table 4.2: Capstone Project Process

Discipline	Activity	Role	Main input artifact	Output artifact
Requirements	Formalize requirements	System engineer	Vision document	Software Requirements Specification (SRS)
	Model specifications	Analyst	SRS	Use case and user interface document
Analysis & design	Analyze use cases	Software engineer	Use case and user interface document	Use case realization
	Define the architecture		Use case realization	Architecture & design document
	Design classes			
Implementation	Implement components	Implementer	Architecture & design document	Components
	Fix defects			
	Implement unit tests			
	Execute unit tests		Components	Unit test results
Testing	Plan tests	Tester	SRS	Test plan
	Describe tests		Test plan	Test cases
	Execute tests		Test cases	Test results
	Evaluate tests		Test results	Change record
Project management	Plan phases and iterations	Project manager	SRS	Iteration plan
	Plan meetings		Work Breakdown Structure	Meeting agenda
Configuration management	Plan configuration management	Configuration manager	Vision document	Configuration management plan
	Manage product configuration		Components	Code management tool repository
Reuse	Find code	Reuser	Component documentation	Validated component document
	Understand code			
	Prototype code			

4.5 Analysis and Results

4.5.1 Time Slip Tokens

The developers of the SFLphone project produced 1930 validated time slip tokens for a workload of 1813 hours. However, tokens related to academic and technical activities were not accounted for in this analysis, since they were not specific to project development. Academic activities are related to the academic course, such as teamwork training and project presentation. Technical

activities are related to tasks which can be performed by technicians, such as configuring the network or setting up and maintaining the development environment. Consequently, 1591 development tokens were retained for an analysis totaling 1560 hours. A coding scheme was defined based on knowledge flow model cognitive factors. For each development token, two independent coders had to decide which cognitive factor was dominant. With a total of 1472 agreements on 1591 judgments, the Perreault & Leigh reliability index was 0.96, which indicates a very strong categorization reliability based on the coding scheme.

4.5.2 Knowledge Flow Analysis

The effort distribution analysis from time slip tokens allows a better understanding of a project's knowledge flow. The SFLphone project had two milestone deadlines. The first (at 40% of project completion) required the development team to present their system architecture to the industrial partner. The second deadline was system delivery at the end of the semester.

Figure 4.3 presents the total effort expended on each cognitive factor in relation to project completion. Each of the 6 curves on the graph represents the total effort expended (Y-axis) for a given cognitive factor with respect to the percentage of project completion (X-axis). For example, at 20% of project completion (X-axis), 7% of the total effort (Y-axis) had been expended on crystallization. Validation and acquisition each represented 5% of total effort, verification and work organization both accounted for 1%, while realization corresponded to 0% of the total effort. The analysis of the slopes of the 6 curves in Figure 3 provides a better understanding of the relationship between cognitive factors throughout the project.

As depicted in Figure 4.3, at the end of the first phase (40% of project completion), crystallization is the most important cognitive factor of the project (19% of the total effort), requiring an almost constant effort, as can be observed from the linear part of the cumulative effort curve. This behavior is typical of disciplined software processes, because many development artifacts are produced in this timeframe, such as the software requirements specification (SRS), a use case document, and an architecture and design document. Validation accounts for 9% of the total effort. At this point, validation effort distribution varies, depending on the crystallized development artifact to review. Acquisition represents 7% of the total effort. This is related to new knowledge needed from developers in order to understand the project's

domain or elaborate the system architecture. Work organization is not a significant cognitive factor, accounting for only 2% of the total effort. However, it is worth noting that this cognitive factor requires an almost constant effort for the duration of the project, as can be demonstrated from the linear part of the cumulative effort curve throughout the project. Realization and verification are unimportant, with both accounting for only 1% of the total effort. The effort invested at that point relates to prototyping and reviewing code for reuse.

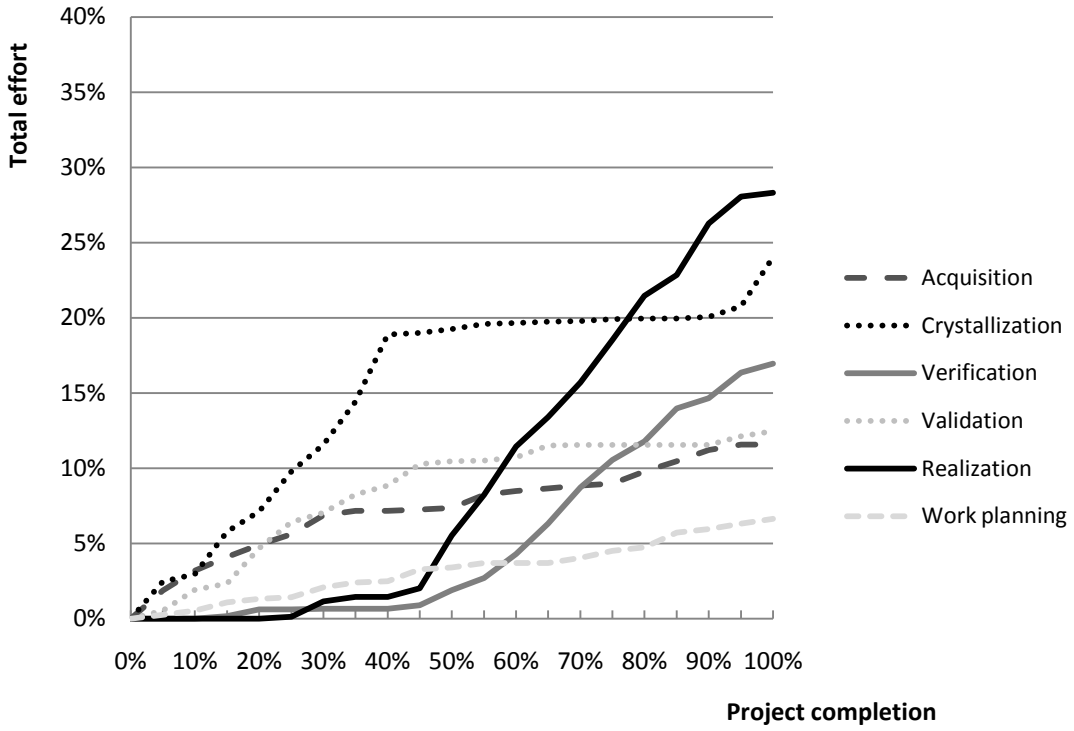

Figure 4.3: Cognitive Factor Effort Distribution in the SFLphone Project

4.5.3 FLOSS Library Issues

The main issues encountered in the SFLphone project were related to FLOSS library documentation: libavcodec, video4linux, and ccRTP.

4.5.3.1 Libavcodec

Several new features of the project required the system to encode and decode video images captured by the webcam. The libavcodec library, a popular open source LGPL-licensed library of

codecs, helped fulfill this requirement. It could be used in two ways: through the command line or as an integrated part of a software project.

Three different strategies were tested to learn how to use the library. First, online documentation was available through the library's website. Second, a user/developer mailing list could be added and used to ask questions. Third, an IRC chat channel made direct interaction possible between users and developers.

The online documentation was mainly used by direct users, who are individuals using libavcodec's encode and decode functions. It was composed of a frequently asked questions (FAQ) section and an online general documentation section providing answers to most of the users' direct questions. The general documentation also covered the development of the open source library, its development policies (coding conventions), and a review process. Developers wanting to use the library as part of their software were invited to refer to a small API code example and look at the source code. If they had more questions, they were encouraged to subscribe to the library's mailing list and participate in chat sessions.

The API code example is a page created by a user willing to share his knowledge with other developers (Böhme, 2004). It shows the main function calls without much explanation of why they were called in that particular order. Moreover, most of the functions in this example were no longer recommended in the source code's comments. The source code only pointed out what new functions should be used without specifying how to use them.

At one point in the SFLphone project, a developer wanted to convert the color format of each pixel from RGB to YUV. The function invoked in the code example was rejected and a new, completely independent library was recommended instead. The documentation was ambiguous, and, consequently, the developer blindly tested the new functions suggested in the source code comments.

As pointed out in the online documentation, a developer could use the mailing list available to developers. Many questions were asked, but most developers did not know the answers. The few experts willing to take the time to respond (mainly the developer in charge of the project) had to answer almost all the questions. Sometimes, they could only speculate on how a given component was supposed to be used since they were not always directly involved in its

development. Moreover, experts capable of answering very specific questions were usually hard to contact, since they were not using the mailing list.

Also, experts commonly suggested searching through the mailing list archive, which was considered by many as the main documentation for developers wanting to integrate the library into their software. This suggestion involved reading each and every existing mailing list thread to find relevant information as the mailing list archive search function was not very effective.

The other method allowing interaction with experts was online chatting, which seemed at first to be the quickest way to find answers. However, only basic questions were answered quickly, and the more difficult ones were left, presumably because the expert on that particular matter was not online. The SFLphone project developers found that browsing the mailing list archive was more effective than asking questions via online chatting, which tended to remain unanswered.

4.5.3.2 Video4linux

Many features of the SFLphone project required video capture from the webcam. The industrial partner strongly suggested using the video4Linux (V4L) library, which is a video capture API now integrated into Linux's kernel. More specifically, V4L is an abstract layer between video software and hardware. It allows both image and video capture from external devices, such as a TV tuner or a webcam.

The industrial partner provided a small, homemade prototype to show the basic usage of the library. The team's strategy regarding reusing V4L consisted of a prototyping strategy. Since every requirement related to the video features obviously involved being able to capture images from the webcam, the team realized that developing a more complete prototype was a top priority, in order to validate required functionalities before integrating V4L into the SFLphone. This strategy was aimed at acquiring better V4L behavioral knowledge before integration, thus minimizing problems. It also focused on technological risk management by allowing a better estimate of the integration effort.

Unfortunately, V4L integration did not occur as planned, mainly because of ambiguous documentation, a problem which was not fixed because of incomplete validation prototyping. For instance, a V4L function was announced as being able to change a webcam's frame per second (FPS) rate. However, the FPS setting was not supported by many webcams. The developers were

unaware of this until they encountered a *segfault* resulting from an incorrect use of the FPS setting function. Of course, this problem could have been avoided if proper prototyping validation had been performed, or if the V4L documentation had been clearer.

4.5.3.3 ccRTP

The Real-time Transport Protocol (RTP) defines a standardized packet format for delivering audio and video over the Internet. The ccRTP library is a C++ implementation of RTP.

The initial version of SFLphone, which required enhancement with video capabilities by the capstone project team, already used the ccRTP library for audio transportation. RTP was designed for both audio and video transportation, and ccRTP documentation seemed to confirm that it supported both transportation types. Consequently, no prototyping effort was invested in ccRTP functionalities. This decision had major consequences for the outcome of the project. It turned out that ccRTP was designed to support video, but had not yet been implemented in the library.

Once again, inaccurate documentation led to false assumptions, which could have been mitigated or avoided if proper prototyping validation had been performed.

4.5.4 Consequences of Ambiguous Library Documentation

The consequences of library documentation ambiguities were analyzed based on the time slip tokens and interviews with the developers that validated our assignment of those tokens. The interviews were conducted on a voluntary basis after the project had been completed and graded.

The documentation for libavcodec and video4linux was partially ambiguous. Specifically, the behavior of both library functionalities was sometimes difficult to predict. In the two cases, the strategy used to fix the problem was black box unit testing, which allowed better understanding of how functions should be invoked.

In the end, libavcodec integration required a total effort of 103 hours. Forty-four hours were considered a normal effort on libavcodec integration. The remaining 59 hours were considered extra effort expended due to inaccurate documentation. Consequently, this problem added substantial effort (134%) to the integration of this library.

Video4linux integration required a total effort of 82 hours. Forty-five hours were considered normal activities, while the remaining 37 hours would not have been required if the documentation had been appropriate. This problem increased the integration effort by 82%.

The ccRTP library issue is linked to the serious consequences of the false assumption that ccRTP supported video transportation. Architectural choices were made based on erroneous assumptions. Since the issue was discovered very late in the project, there was not enough time left to implement another solution, although this possibility was considered. The only viable solution was to downsize the project by not implementing all the functionalities.

Nevertheless, a great deal of effort was expended in reaching the conclusion that the documentation was not up to date. ccRTP integration required a total effort of 132 hours. Only 24 hours were considered normal activities, while the remaining 108 hours would not have been required with appropriate understanding of the library functionalities. This problem increased the integration effort of the library by 450%.

Table 4.3 summarizes the normal and extra effort expended on libavcodec, video4linux, and ccRTP library integration. The three documentation-related issues increased the total project effort by at least 204 hours, representing the equivalent of 5 full-time man-weeks.

This quantitative evaluation represents only one aspect of the repercussions of ambiguous documentation. Another major drawback is the quality of the delivered product, which can be difficult to evaluate.

Table 4.3: Component effort distribution

Component	Normal (h)	Extra (h)	Extra added (%)
Libavcodec	44	59	134
V4L	45	37	82
ccRTP	24	108	450
Total	**113**	**204**	**181**

4.6 Discussion

Since the project was developed as a capstone project, the schedule was non negotiable. Whatever happened, the development team had no choice but to deliver a functional product after 14 weeks of development. Both the schedule and the human resources (5 developers) were fixed,

so the only flexibility left was with the product functionalities. The industrial partner and the development team agreed that it was preferable to cut down on these. Consequently, the requirements related to the three-way audio and video conferencing features were not implemented as part of this project. However, other requirements were successfully developed.

4.6.1 Extra Effort Distribution

The following figures (4.4, 4.5, and 4.6) present quantitative analysis of the cognitive factors (acquisition, verification, and realization) affected by the documentation issues of FLOSS components (libavcodec, video4linux, and ccRTP). The crystallization, validation, and work organization cognitive factors were not measurably affected by the documentation issues.

Figure 4.4 shows the cumulative effort expended on the acquisition cognitive factor during the real project (dark line) and the adjusted curve (light line), and represents the expected effort given appropriate documentation. The revised curve was obtained by removing from the time slip tokens all entries related to the extra effort required by the documentation problems, based on interviews with two of the project's developers.

The real acquisition cognitive factor effort increased around mid-project (60% towards the end), which means that unusual effort was expended after mid-project to acquire new knowledge. This indicates that something went wrong, as it is unusual after mid-project to expend any significant effort on the acquisition of new knowledge. The adjusted curve is representative of measurements on successful projects.

Figure 4.5 shows the real and adjusted verification cognitive factor cumulative effort. The verification cognitive factor involves mainly test-related activities. Usually, this curve is almost linear from mid-project to the end of the project, which can be observed as a constant slope for the project duration, which is the behavior represented by the adjusted curve. The SFLphone project ("real" curve) put an extra burden on the verification activities, as observed by the increase in the slope starting at 55% of project completion.

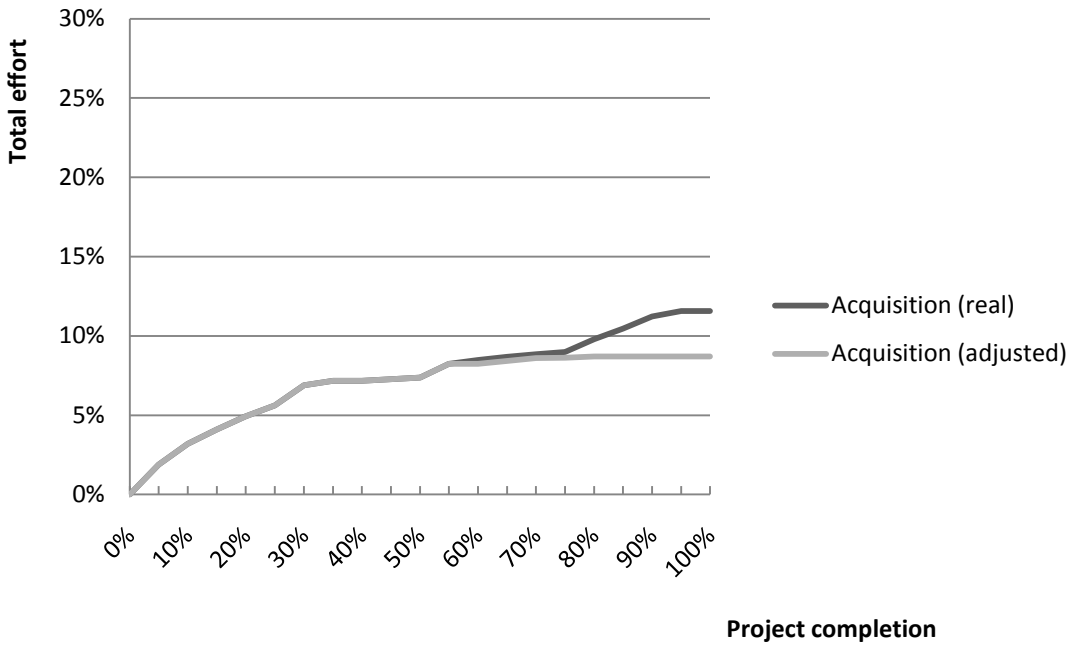

Figure 4.4: Real and Adjusted Acquisition Cognitive Factor Cumulative Effort

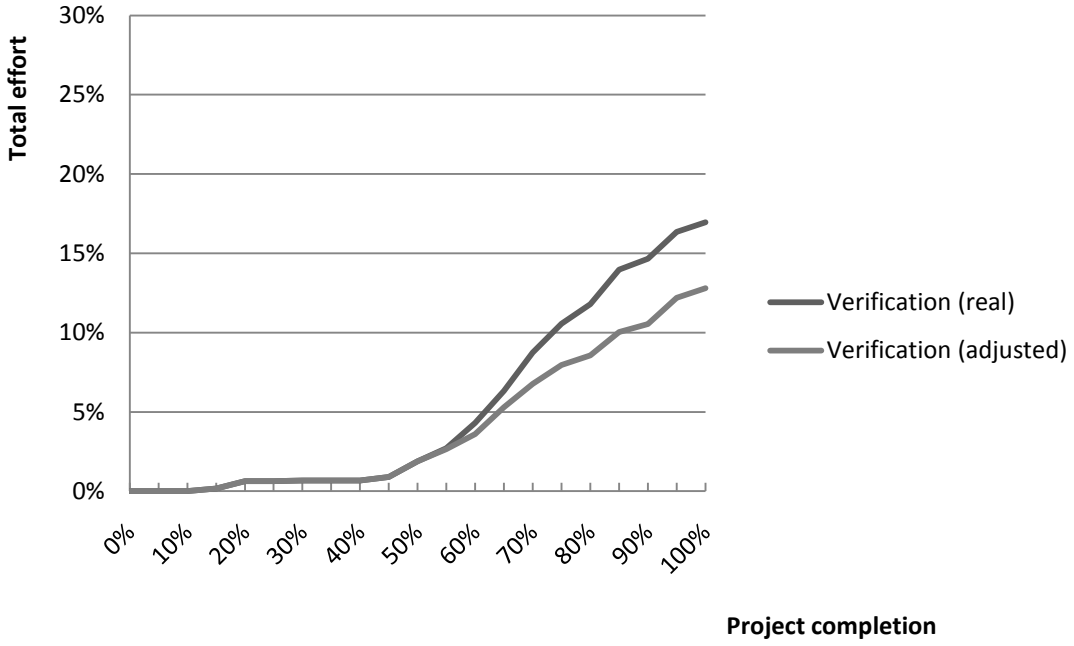

Figure 4.5: Real and Adjusted Verification Cognitive Factor Cumulative Effort

Figure 4.6 shows that extra effort on the realization cognitive activities started at the 55% point of the coding effort. A strong level of coding activities was maintained until the very end of the project. The adjusted curve of the coding activity shows a smoothing at 80% of project completion, which is observed in successful projects.

These curves provide only a partial picture of the extra effort attributable to the documentation issues. Unimplemented functionalities required a significant effort expended on design, test plans, test cases, and meetings with the industrial partner, which were not taken into consideration in this analysis.

Moreover, the team's disappointment when they realized that the desired product could not be delivered, as well as the client's frustration when they received a different product than expected, were not incorporated into this analysis.

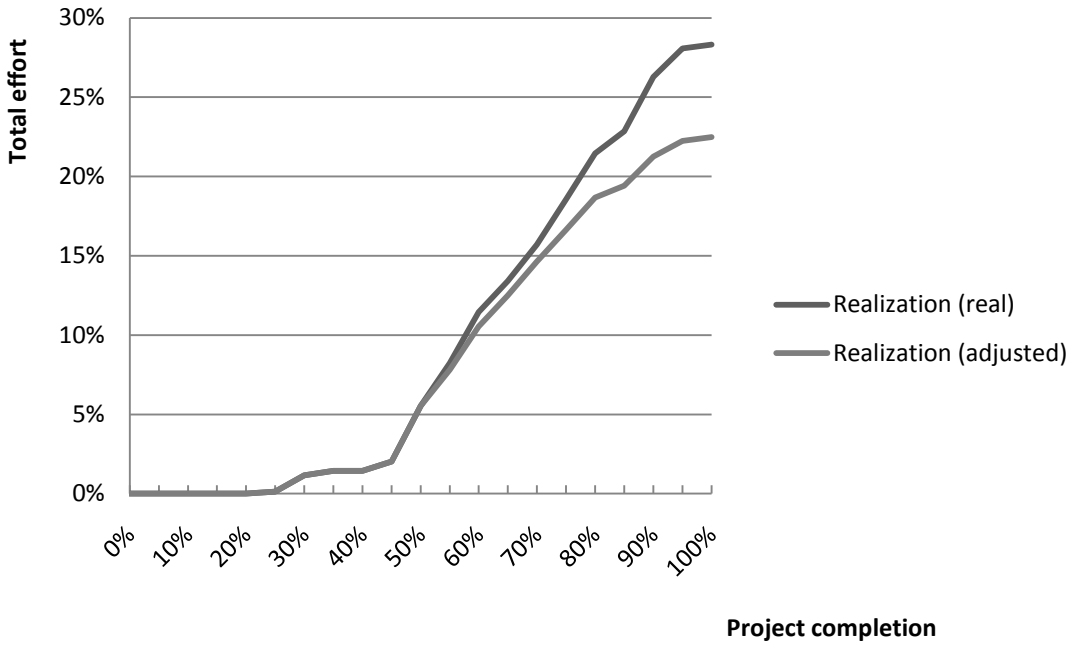

Figure 4.6: Real and Adjusted Realization Cognitive Factor Cumulative Effort

4.6.2 Software Practice Recommendation

The SFLphone project helped us to understand that, in free/libre open source applications, considerable effort is expended on software development, but not so much on documentation.

To maximize reuse benefits, we recommend adding a stronger process practice enforcing reusable code validation and including a feedback loop.

Figure 4.7 details the reusable code validation practice. The reusable component is used by the reuser as the input artifact. The reuser is responsible for validating the component's reusability. This activity produces a reusability report as the output artifact. Then, a reviewer will revise the reusability report to ensure that it is reliable. The revised reusability report will be sent to the reuser, if needed. This strong feedback loop will prevent improper enforcement of component reusability validation, as was the case in the SFLphone project.

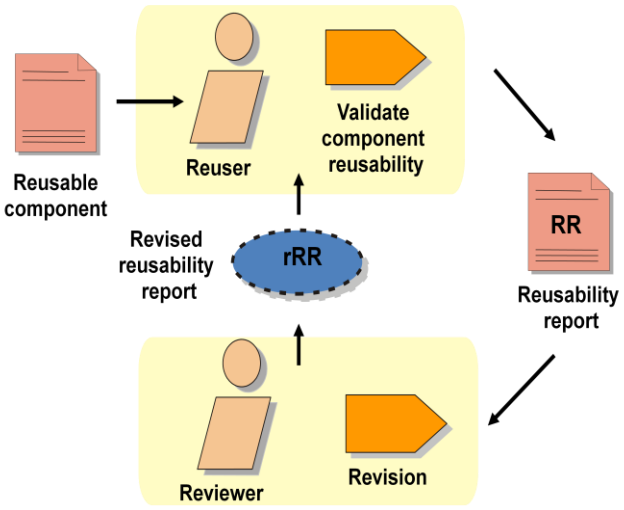

Figure 4.7: Reusable Code Validation Practice

The reusable code validation practice must be completed prior to architectural and design activities, because important information will come out of the practice which could influence architectural and design decisions. Moreover, this practice will help in managing technological risk, because a better understanding of a component's functionalities will facilitate subsequent effort estimation.

4.7 Conclusions

The reuse of free/libre open source software components is a recommended strategy. However, it is important to understand the possible consequences of such a choice. Based on the literature and the results of this case study, FLOSS documentation appears ambiguous, imprecise, and out of

date. This leads to effort being expended fixing preventable issues. In the case of the SFLphone project, at least 200 hours were lost due to the ambiguity of FLOSS documentation. We suggest that the enforcement of a reusable code validation practice would lower free/libre open source component reuse effort.

4.8 Acknowledgments

The authors wish to express their gratitude to Savoir-Faire Linux for their participation in the 2008 edition of the Capstone Project. We extend our special thanks to Benoit Grégoire, the on-site engineer who provided assistance to the development team once a week. We also thank Jean-François Blanchard-Dionne, one of the student developers, who provided insightful information regarding the development of the project.

CHAPITRE 5

IS DESIGN USEFUL IN SMALL SOFTWARE PROJECTS? AN EXPLORATORY CASE STUDY

5.1 Abstract

It is generally accepted that there are discrepancies between design artifacts and implementation. These discrepancies have been studied from the design erosion or software evolution perspectives. Many approaches have been proposed to retrofit or reverse engineer the source code *in lieu* of design. Other approaches recognize the discrepancies and focus on keeping the best of design and implementation. Other approaches insure information continuity from the design to the source code. The purpose of this study is to understand the mechanisms leading to discrepancies by analyzing a case study of a software development project based on a disciplined software development process. Unlike design in traditional engineering where the blueprint artifacts are abstract models of products to be implemented, software design artifacts are rather images of product possibilities. We found that discrepancies between the designed and the implemented classes are a consequence of the implementation activities rather than the result of a design evolution.

5.2 Introduction

Design activity is a major component of any software process approaches. It is generally accepted that there are discrepancies between design artifacts and resulting implementations. Many works have been interested in finding relationship between design and code. This paper analyses the emergence of discrepancies between design activities and resulting implementation. This work sheds some light on this aspect by analyzing the data collected during an industrial software development project.

Some authors call design erosion the drift between design and implementation (van Gurp & Bosch, 2002). They have found that software designs tend to erode over time to the point that redesigning from scratch becomes a viable alternative compared to prolonging the life of the existing design. They demonstrate that design erosion is inevitable because of the way software is

developed. They have found evidence of architectural drift, vaporized design decisions and design erosion. An important conclusion is that even an optimal design strategy for the design phase does not deliver an optimal design.

Others prefer to call it software evolution (Li, Etzkorn, Davis, & Talburt, 2000). In this study, three metrics—System Design Instability, Class Implementation Instability, and System Implementation Instability—are used for the purpose of measuring object-oriented (OO) software evolution. The metrics are used to track the evolution of an OO system in an empirical study. They performed a study of design instability that examines how the implementation of a class can affect its design. This study determines that some aspects of OO design are independent of implementation, while other aspects are dependent on implementation.

Many researchers have proposed to bridge the gap between design artifacts and implementation by reverse engineering the implementation into design artifacts. The software engineer might use a reverse engineering system to derive a high-level model from the source code. These derived models are useful because they are, by their very nature, accurate representations of the source. Although accurate, the models created by these reverse engineering systems may differ from the models sketched by engineers.

Many studies have investigated the relationships between these so-called design metrics and product quality in terms of fault-proneness (Yuming & Hareton, 2006; Briand, Wu, & Lounis, 2001). Other studies have used OO design metrics to evaluate the testability of the resulting designed software units (Subramanyan & Krisnan, 2003). Almost all facets of OO software have been analyzed in terms of OO design metrics, such as predicting effort (Baudrya & Le Traon, 2005) and predicting maintenance (Alshayeb & Li, 2003). Some researchers, however, are questioning the prediction capabilities of OO metrics and their usefulness (Fioravanto & Nesi, 2001). Most of these studies capture the results yielded by what their authors call design metrics. In fact, it is the modeled representation of the design of the implemented classes that they are capturing, those classes being retrofitted into UML representations.

For many developers the original design artifacts are important and effort must be made to bridge the gap between design and implementation. The software reflection model technique permits an engineer to summarize structural information extracted from the source within the framework of a high-level model (Murphy, Notkin, & Sullivan, 2001). It provides a means of bridging the gap

between the high-level models commonly used by engineers to reason about a software system and the system artifacts that are the software system.

Another approach is to check the compliance of OO design with respect to source code (Antoniol, Caprile, Potrich, & Tonella, 2000). They recover an "as is" design from the code and compare the recovered design with the actual design. Verification of the design-code compliance is the basic step to produce an updated version of a design. Since the design represents an abstraction of the implementation, relations between classes in the design are expected to be all present in the code, while additional relations in the code can be regarded as implementation details. Our study shows that it is not as simple.

Another approach is to insure information continuity. Each software artifact along the software development cycle should be the refinement of the artifacts of the previous phase. It should be consistent with the previous artifact and it should be possible to trace information along the phased development process. Many approaches try to limit the discontinuity of information across different models used in the software process.

A model connectors approach is proposed as ways for bridging information across models in the software life cycle (Medvidovic, Grunbacher, Eegyed, & Boehm, 2003). In particular, they have devised a set of techniques for bridging different design models, both at the same level and across levels of abstraction. Connectors between models satisfy two primary goals: to transform model information or to compare model information.

Another paper describes an integrated traceable software development approach in the context of a use case design methodology (Kim & Carlson, 2001). The foundation for these approach lies in partitioning the design schemata into a layered architecture of functional components called design units. The proposed code generation technique focuses on creating a skeletal code framework at the design stage to get control over the quality of the code and establish a manageable relationship between design and the actual implementation. The proposed design concept provides a framework for supporting traceability through the software development lifecycle. There are many software tools that tend to preserve the continuity between the class designs and the implementation.

The purpose of this study is to understand the emergence of discrepancies between the software design activities in a plan-driven software development process and its implementation. Section 2

describes the industry-based software project on which this study is based. Section 3 presents a model for the cognitive activities involved in the design process. Section 4 discusses the finding of this research and its validity.

5.3 Description of the Project

The requirements for the project came from a collaborating avionics industry. The objective of the project was to build a software tool for editing and configuring systems based on ACS (Avionics Core Software) architecture. The ACS Library is a set of C++ classes, which are the building blocks for developing large-scale avionics systems. The functionalities of the new tool had to be based on class structures taken from the ACS libraries, and had to improve the productivity associated with building systems based on this library. This new software tool also had to be capable of integration into existing tools.

This project was carried out by senior (fourth-year) students in the Studio course in software engineering at the École Polytechnique de Montréal, which is an elective capstone, project-oriented course offered during the last term. Teams of Studio course students must follow a prescribed plan-driven software engineering process. Previous empirical studies on Studio activities concerned such topics as cognitive activities (Germain & Robillard, 2003) and comparing the strengths of engineering-based processes to those of the agile methodologies (Germain & Robillard, 2005).

The Unified Process for EDUcation (UPEDU) (Robillard et al., 2003), is an adapted version of the Rational Unified Process (RUP) (Kruchten, 2000). Its objective is to define a software process which is appropriate to the project and only the activities and artifacts relevant to the targeted projects are retained. The software process for the Studio is well defined, and all the activities and artifact templates can be viewed on the UPEDU website.

All data used for this observational study were collected through online effort slips filled out by participants. The data collection scheme was adapted from a core framework which included the following data elements for each effort slip:

- Participant ID
- Date
- Activity performed (one short sentence in free format)
- Input and output artifacts

- Effort expended (with a 45 minutes granularity)

The participants had been trained to fill out the effort slips correctly, and the slips were checked regularly by the instructors in order to ensure their validity. Ethical issues were handled according to Canadian policy for research involving humans (NSERC, 2005).

This analysis takes into account only the effort that is relevant to software process activities. Examples of effort not considered are the effort expended on training, on setting up the development environment and on preparing the project presentations for design and product deliveries.

The parameters of the 14-week project are the following: The students form a team of five members. They are assigned a dedicated room and are required to spend a day and a half of teamwork on the project. They are coached by an expert from the industry, who visits them once a week, and by an instructor from the school, who visits them often during their weekly working sessions to coach them on software process issues. The instructor is not involved in the resolution of any design or coding issues. The team is required to present a design of the new product to the client before starting the programming activities. In the case reported here, the client was satisfied with the design presented and stated that it was consistent with the level of design expertise in his organization. The effort expended exclusively on the design activities accounts for 10% of the total effort.

5.3.1 Class Categories

The software product delivered by the Studio students is made up of 163 classes. This product, like most real industrial software products, was not built from scratch. The project involved adding functionalities to an existing software product. The 163 classes fall into 6 categories of design activities:

1. The *reused* category contains 121 classes reused from an existing library.
2. The *modified* category contains 6 classes that were partially reused, since they were modified during the implementation phase without being documented during the design phase.
3. The *adapted* category contains 9 classes that were designed to be partially reused.

4. The *added* category contains 5 new classes that were designed specifically for the product.
5. The *created* category contains 22 new classes that were created during the implementation phase without any previous design.
6. The *deleted* category contains 3 classes that were designed, but were not implemented.

Figure 5.1 presents a model of the class categories. The box on the right hand-side of the figure represents the implemented product made up of the 163 classes. The box at the top represents the library, which is the source of the 136 classes (121 reused + 6 modified + 9 adapted), and the illuminated cloud at the bottom represents the mind resource for the 27 new added (5) or created (22) classes. Design activities represented by the broken line box in the middle were involved in the 14 adapted (9) or added (5) classes.

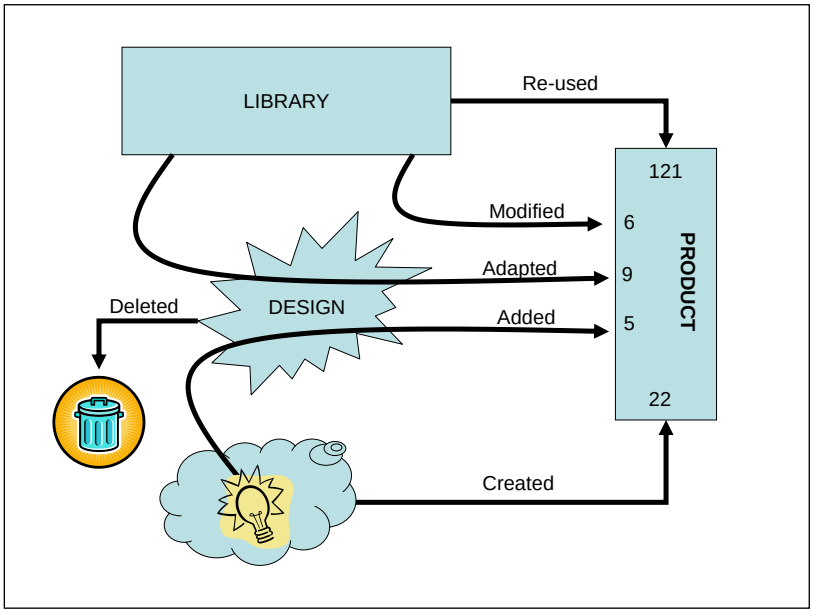

Figure 5.1: Model of class origin

On a class basis, the design activities take into account only 14 classes out of the 42 implemented classes, which accounts for little more than 30% of the product's implemented classes.

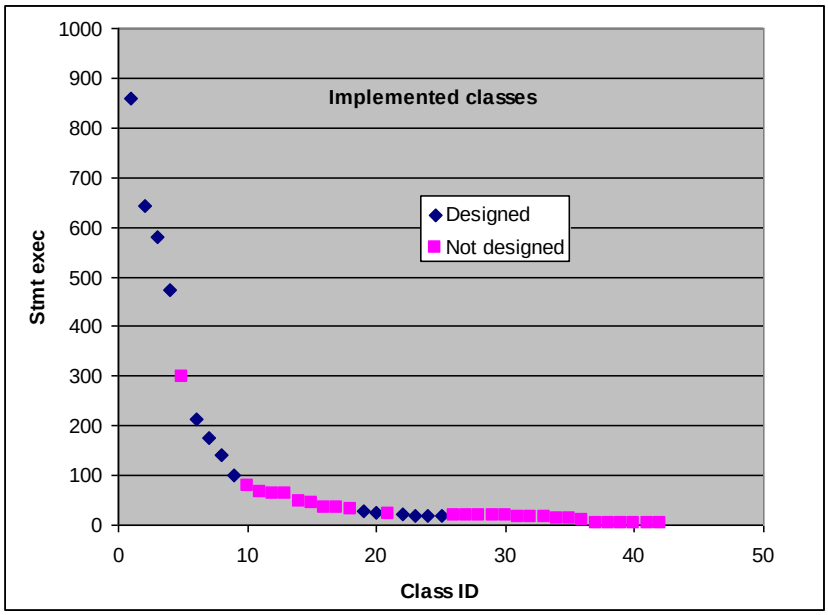

Figure 5.2: Size in number of executable statements of the implemented classes

The class perspective must be complemented with the size perspective. Figure 5.2 shows the size in number of executable statements for each of the implemented classes in decreasing order of number of executable statements. The largest class contains more than 850 executable statements, while almost 60% of the classes have fewer than 30 executable statements. The designed classes account for almost 80% of the executable statements.

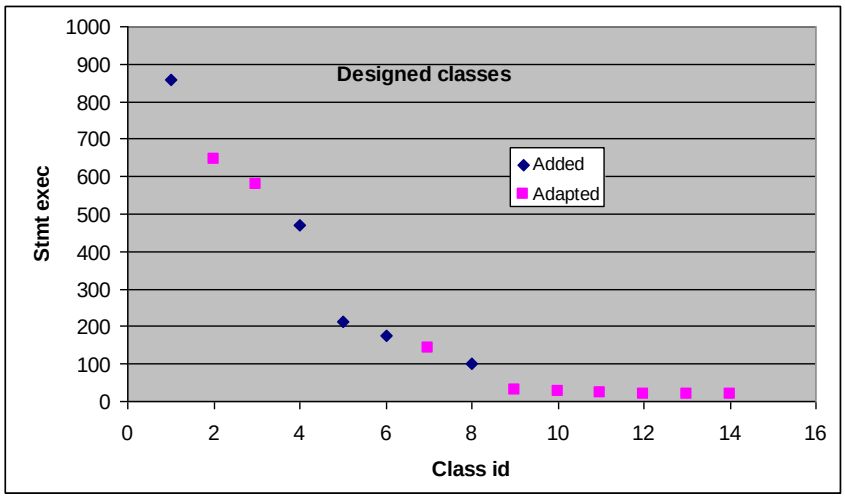

Figure 5.3: Size in number of executable statements of the designed classes that have been added and adapted

Figure 5.3 shows that the design activities are concentrated in large classes that have been adapted from existing classes or in the design of new added classes. A static analysis of the source code reveals a very high correlation between the number of executable statements and cyclomatic complexity.

Figure 5.4 presents the model of the product structure in terms of total size and number of classes percentage for the four categories of implemented classes. It is observed that more than 75% (35% + 42%) of the total number of executable statements involves design activities, and that these account for only a third (12% + 22%) of the total number of classes. It is important to stress that these statements were not designed, but they implement classes that have been designed. A more detailed analysis shows, for example, that some features must have been designed, although we cannot find any corresponding classes in the design artifact and that some features were designed, but implemented in different classes than the one designed. These are cases of mismatches between designed modifications and class implementations.

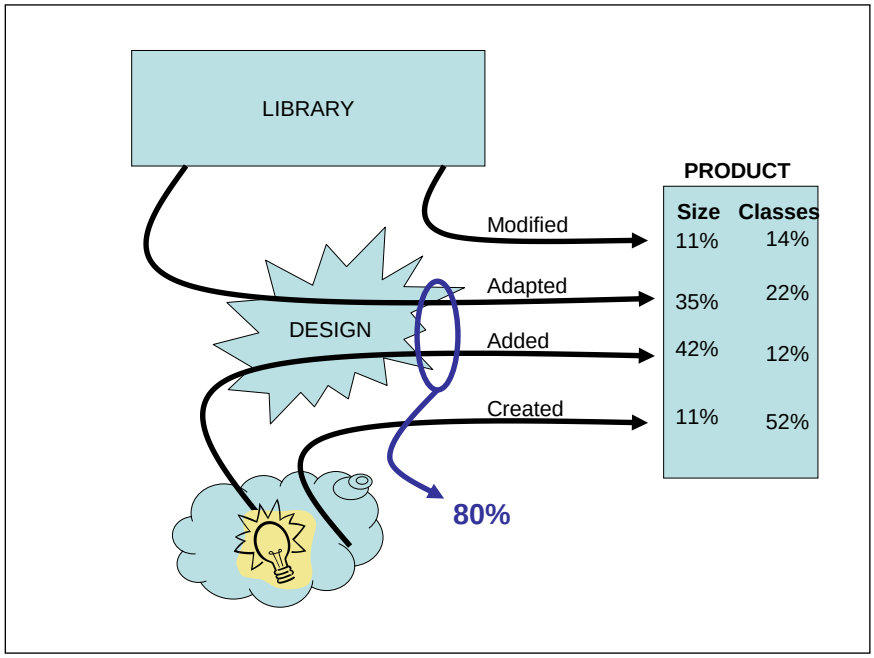

Figure 5.4: Product perspectives in terms of executable statements and number of classes for the designed classes and the classes not designed

We found that these class discrepancies between the designed and the implemented classes are a consequence of the implementation activities rather than the result of a design evolution. Design analysis based only on the implemented class names could be misleading.

5.4 Design Process Activities

In software engineering, design refers to a process discipline, or series of activities. Software design artifacts are different from the technical blueprints found in the civil or mechanical engineering, which must be strictly followed. Software engineering design activities address how the system will accomplish the functional requirements, and these include algorithms, input/output formats, interface descriptions and data definitions. Defining architecture is part of the design process.

Software design is found to be a cognitive opportunistic process leading to the crystallization of an entity, which is the "image of possibility".

Cognitive activities refer to the mental process by which knowledge is acquired and recorded. A set of cognitive actions is *opportunistic* when we must explore further and find missing information to complete the task. Knowledge is partially and incrementally gathered as opportunities present themselves, which in turn depend on the cognitive availability of the necessary material. This is not a well-planned process (Robillard, 2005).

Figure 5.5 presents the relationship between cognitive activities and knowledge sources in software development. This knowledge flow model is related to Nonaka & Takeuchi's (1995) knowledge creation model.

The four rounded-corner boxes in Figure 5.5 represent knowledge sources. External information can be general or specific to the project under development. General information may come from various sources, such as the Web, a paper, or a book. Specific information comes from any of the project's pre-existing documentation. A development artifact is a physical representation of knowledge, such as a software requirement specification (SRS), a design specification, or a test plan. Source code strictly includes executable statements and comments. Tacit knowledge is individual knowledge built from interacting with other knowledge sources.

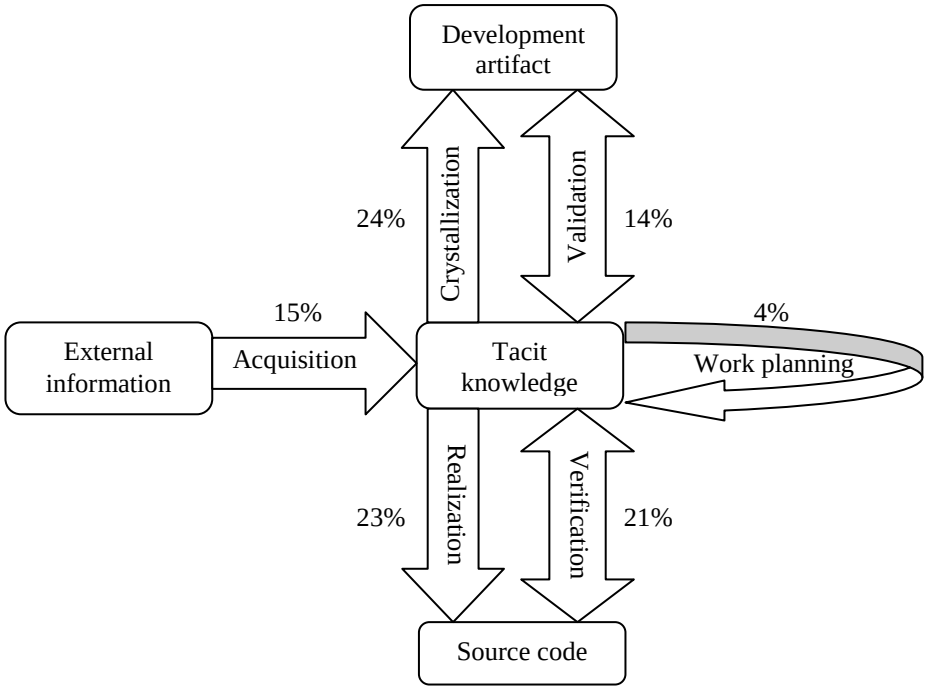

Figure 5.5: Knowledge flow model

The arrows in Figure 5.5 represent the cognitive activities and the percentage next to an arrow represents the effort expended in a cognitive activity relatively to the project's total effort. The acquisition cognitive activity (15% of total effort) is involved when a developer needs to increase his tacit knowledge from external information. The crystallization cognitive activity (24% of total effort) is the translation of a developer’s mental representation of a concept (tacit knowledge) into an artifact (explicit knowledge), such as a use-case diagram or an architectural plan. Crystallization, by means of design artifact, is the representation of the "image of possibility".

Without this representation, the original "image of possibility" becomes an undefined artifact, and in time can vanish altogether. This is not to say that the crystallized image does not change during the subsequent implementation process, for it does, and often quite drastically. The realization cognitive activity (23% of total effort) also involves the translation of tacit knowledge into explicit knowledge, but requires, in addition, technical know-how, which is related to source code production. The validation cognitive activity (14% of total effort) involves bidirectional knowledge flow between tacit knowledge and development artifacts (explicit knowledge), in order to validate the consistency of those two knowledge sources. The verification cognitive activity (21% of total effort) is like validation, except that source code is the knowledge source,

thus involving technical know-how. The work planning cognitive activity (4% of total effort) mostly involves developers' synchronization of the project's planning and progress knowledge.

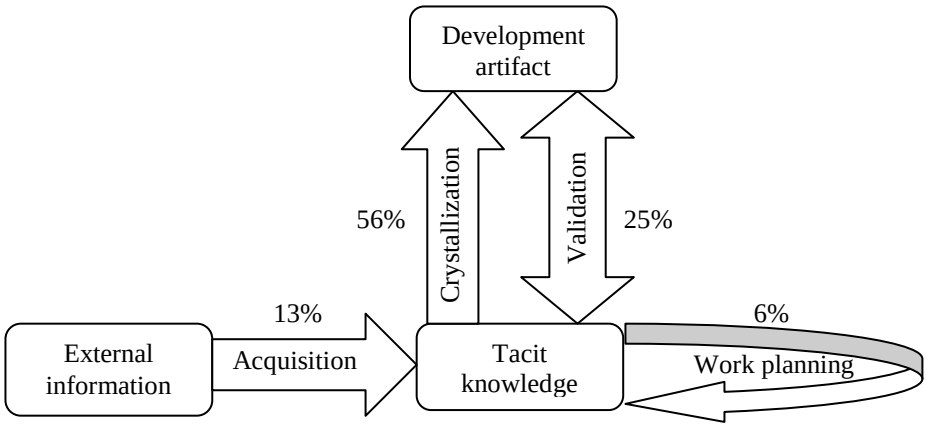

Figure 5.6: Model of cognitive activities performed during the design process

Figure 5.6 illustrates the cognitive activities specific to the design process as observed in this exploratory case study. Acquisition activities, which account for 13% of the design effort, mainly involve reading and searching on the Web for information that is relevant for the design. Crystallization activities, which account for 56% of the design effort, involve diagramming (mostly with the software tool *Together Control Center*) and producing the design artifacts. Validation activities, which account for 25% of the design effort, involve review and inspection of the design artifacts. Work planning activities, which account for 6% of the design effort, involve design planning and progress tracking. All retrofitting or refactoring activities based on the implemented classes are excluded from the design activities category.

The design phase is mainly collaborative (56% of the design effort), unlike other project development phases (requirements, implementation, tests), with 39% of the total effort being collaborative.

5.5 Discussion

The real design process is composed of activities performed following requirements analysis and prior to product implementation. This study is based on a design environment, in which software engineers perform some design activities after the requirements phase and before the implementations phase in a real project. Some software processes will make design activities

more explicit than others. Professional software design is inherently complex to study for the following reasons:

- It is mainly a cognitive activity which cannot be measured directly;
- It is performed by team of developers, and it is difficult to know who is doing what;
- The activity boundaries between requirements, design and implementation are fuzzy;
- Design artifacts are often not implemented as designed, and some implemented artifacts have not been designed.
- There is a wide variability in design environments and constraints regarding the nature of the application, the team's experience and the project parameters.

For all these reasons and many others, any study based on a case study is one of a kind. Data values are only indicative of one very specific case. However, many successful software products are being made by teams of developers involved in some design activity. We believe that studies like this one are important because they shed light on an essential process which might not differ fundamentally from team to team.

The whole process of developing software is a complex endeavor because it involves mainly cognitive activities, and the final product is a set of instructions to enable a computer to implement some functionality. There are four major cognitive steps that must be performed by the team of developers at the project level:

1. to understand the requirements,
2. to understand the product to be developed,
3. to implement the code instructions, and
4. to test the resulting computer program.

These cognitive steps are combined and performed concurrently to varying degrees, depending on the software process adopted. The design process observed in this case study is essentially the second cognitive step, which is to understand the product to be developed.

This exploratory case study provides some insight into the design process. The various values obtained from the measurements based on this case study quantify to some extent the characteristics of the design process, which are summarized below.

Designers concentrate on the major features to be implemented, which are grouped together in a few software units with little concern as to the size of the software unit. The important concern is the relationships among the various features. The design effort, which is the effort expended to understand the product to be implemented, is found to be of the same order of magnitude as the effort expended to understand the requirements. Excluding the deleted and reused classes, four class categories are involved in software projects:

1. classes created during the implementation phase (without design) because they could not be foreseen during the design phase or are small utility classes;
2. classes added as designed;
3. reused classes redesigned to be adapted for the new applications; and
4. reused classes modified during the implementation phase without any previous design.

The design process, which concentrates on the added and adapted classes, targets most (75%) of the implemented executable statements.

The design process is made up of four major cognitive activities, which are:

1. the acquisition of knowledge,
2. the processing of this knowledge by exchanging information with teammates,
3. the crystallization of this knowledge on appropriate design artifacts, and,
4. the validation of the crystallized knowledge by inspecting the artifacts.

The process is opportunistic and depends on the knowledge available at any given time.

Design activities have two purposes. One is to provide design artifacts and the other is to synchronize the mental models of the developers on the product to be implemented. Design artifacts were not realized as blueprint in this case study. They were rather the crystallization with the UML notation of an incomplete image of the product. These artifacts were used to improve teammates understanding of the product to be implemented.

In this study, we found the design process to be most useful as a set of cognitive activities performed to enable the understanding of the product to be implemented.

Regardless of the software process, disciplined or agile, teammates need to synchronize their mental image of the product to be developed. In order for teammates to do so on the same mental model, some design artifacts are needed to crystallize ideas in the mind. Starting the

implementation phase by ensuring that all teammates have the same image of the product in mind reduces the risks of worthless implementation effort.

This study has led to two major observations that have an impact on any software process paradigm.

OBSERVATION A: Design is shared learning.

The whole design process is found to be a mechanism for cognitive synchronization of the teammates on their understanding of the perceived product. Indeed, more than half (56%) of the design effort is expended in collaborative activities (crystallization, validation and work planning). These activities are opportunistic in the sense that they occur in a just-in-time or on an as-needed basis.

OBSERVATION B: Software design constitutes the elaboration of an "image of possibility".

This study shows that design is not performed as in the traditional engineering process. Software design artifacts constitute an intermediate crystallization of a model of the product, or, more poetically, an "image of possibility".

Discrepancies between design artifacts and implementation can be characterized as either undocumented design improvements or redesign related to better requirements understanding.

In the analyzed project, the three-tier MVC (Model-View-Controller) architectural pattern was employed to isolate business logic from user interface. For the View tier, discrepancies are related to minor design improvements. For the Controller tier, discrepancies are related to significant design improvements, especially to the most significant class of the project (853 executable statements). Significant discrepancies reveal that designed "images of possibility" were not adequate for implementation. For the Model tier, 17 classes were created during implementation, without prior design. However, the changes are related to minor redesign due to better understanding of requirements and to design optimization. Even though a significant number of classed were created during implementation, only 443 new executable statements were added. Moreover, we observed that developers will not design small classes, and that the big design classes will be implemented more conveniently in smaller classes. We believe that this behavior is acceptable because the team mentally constructed more or less the same model during the design activity.

5.6 Conclusion

This study explores the design process with respect to the classes, the executable statements and the activities performed. Data were captured from an industrial project realized by a team of five senior students as part of their capstone project. The resulting software product was of good quality, and has been integrated into the participating industry's software environment. The project was successful and all the milestones were achieved on schedule. One of these was to deliver and present the design of the product before the implementation iterations began.

We make two observations, which are that design activities are primarily used to share the learning on the product to be implemented, and that software design artifacts, unlike traditional engineering artifacts, are actually images of possibilities.

It is reasonable to believe that these observations may hold for a certain number of software projects which are similar to the one addressed in this case study.

Studies with students can be criticized on the basis of their degree of external validity, and this is a subject which has been discussed in the literature. The degree of validity of this study has been increased by relying on senior students enrolled in their last semester and who have had some internship experience in industry. The line between these students and novice professionals is becoming blurred. From studies that have been conducted to evaluate the difference between software engineering students and the professional software developers used as subjects in empirical studies, it has been found that the differences are only minor. It has, in fact, been concluded that software engineering students may substitute for professional software developers under certain conditions (Carver et al., 2003).

Research areas that flow from this study can be divided into two categories. The first relates to the generality of the findings of this case study. The impacts of our study regarding design activities should be validated in a variety of industrial settings. The second relates to process activities and the expected content of a design artifact.

5.7 Acknowledgments

We are grateful to CAE Inc. for their involvement in this project. Thanks to Jean-François Campeau for his dedicated coaching of the students. This project was made possible by the

participation of the five students enrolled in the Studio in software engineering. Thanks to François Kemp, Robert Morin and Walid Bouzouita, who participated in the data analysis. This project was supported in part by grant NSERC A-0141.

CHAPITRE 6

RÉSULTATS COMPLÉMENTAIRES

6.1 Introduction

L'objectif de ce chapitre est de présenter des résultats complémentaires permettant de caractériser les projets intégrateurs finaux en génie logiciel C6, C7 et C8. Cette caractérisation repose sur la méthodologie ATS et la modélisation par flux de connaissances (section 6.2). D'abord, les caractéristiques générales des projets seront présentées (section 6.3). Par la suite, les développeurs seront caractérisés par l'analyse de leur production de jetons (section 6.4). Puis, l'effort sera caractérisé sous plusieurs perspectives (section 6.5). Finalement, une discussion présentera les contributions issues de ce chapitre (section 6.6).

6.2 Modèle de flux de connaissances

Afin de permettre la caractérisation de projets logiciels selon une perspective de flux de connaissances, un modèle de flux de connaissances a été développé selon l'approche de théorie à base empirique (*grounded theory*). Selon cette approche, un modèle est élaboré à partir de données, jusqu'à l'atteinte d'une saturation conceptuelle, c'est-à-dire que le modèle représente adéquatement les concepts observés à partir des données. Ces données proviennent des jetons ATS consignés par les développeurs des projets intégrateurs entre 2006 et 2009. Un échantillon des ces jetons est présenté à l'annexe C.

Avant de parvenir à la saturation conceptuelle, plusieurs itérations du modèle ont été nécessaires. D'abord, des "types de connaissances" ont été proposés dans un article de conférence SEKE'07 (annexe A). Ces "types de connaissances" s'inspirent du modèle organisationnel de création de connaissances de Nonaka et Takeuchi (1995). Ainsi, 8 types de connaissances sont définis pour représenter le flux de connaissances en développement logiciel: la conversion collaborative tacite-tacite (CTT), la conversion tacite-explicite (TE), la conversion collaborative tacite-explicite (CTE), la conversion explicite-explicite (EE), la conversion collaborative explicite-explicite

(CEE), la conversion explicite-tacite (ET), la conversion collaborative explicite-tacite (CET) et le savoir-faire (KH).

Par la suite, l'article de conférence eKNOW'09 (annexe B) regroupe les 8 types de connaissances présentés précédemment en 5 facteurs cognitifs pour représenter le flux de connaissances dans un projet logiciel: synchronisation (conversion TT), cristallisation (conversion TE d'artefacts), réalisation (conversion TE de code source), acquisition (conversion ET) et validation (conversion EE). En fait, pour des fins de simplification, ce modèle ne différencie pas les activités cognitives collaboratives des activités cognitives individuelles. De plus, une différenciation de conversion TE est introduite entre la cristallisation d'artefacts et la réalisation de code source.

Puis, un autre modèle de flux de connaissances a été développé et présenté dans l'article soumis à la revue *Empirical Software Engineering* (chapitre 3). Ce modèle définit désormais 6 facteurs cognitifs: cristallisation (conversion TE d'artefacts), réalisation (conversion TE de code source), validation (conversion EE d'artefacts), vérification (conversion EE de code source), acquisition (conversion ET) et organisation du travail. Comparativement à la version précédente du modèle, la synchronisation est répartie dans les autres facteurs cognitifs, la vérification de code source est introduite afin de la différencier de la validation d'artefacts et l'organisation du travail est ajoutée, pour différencier les activités cognitives relatives à la planification et à l'avancement du projet.

Finalement, la dernière itération du modèle de flux de connaissances, présentée à la figure 6.1, met l'accent sur les connaissances tacites individuelles, qui sont centrales au modèle de flux de connaissances. Le facteur cognitif d'organisation du travail a été supprimé puisqu'il n'est pas lié strictement au développement logiciel, mais plutôt à la gestion de projet. Ainsi, le modèle représente les possibilités de conversion de connaissances explicites et tacites au sein d'un projet de développement logiciel sur une base individuelle.

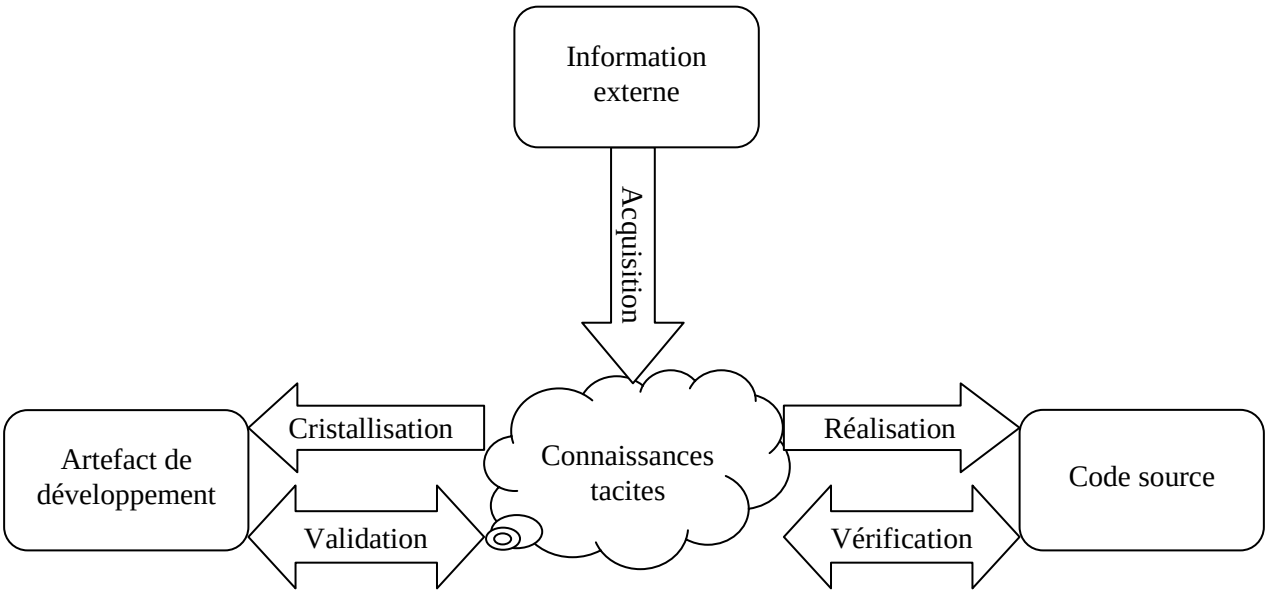

Figure 6.1 : Modèle du flux de connaissances d'un développeur logiciel

Les trois boîtes aux coins arrondis de la figure 6.1 représentent des sources de connaissances explicites, alors que le nuage au centre représente les connaissances tacites d'un développeur. L'information externe peut être générale ou spécifique au projet à développer. L'information générale peut provenir de diverses sources telles que le Web, un article ou un manuel. L'information spécifique provient de toute documentation préexistante du projet. Un artefact de développement est une représentation physique de connaissances telles qu'un SRS, ou un plan de tests. Le code source n'inclut que les déclarations (*statements*) et les lignes de commentaires. Les connaissances tacites sont construites par l'interaction entre un développeur et des sources d'information ou d'autres développeurs.

Les flèches de la figure 6.1 représentent les facteurs cognitifs, qui constituent le flux de connaissances entre les sources de connaissances explicites et les connaissances tacites. L'acquisition survient lorsqu'un développeur doit augmenter ses connaissances tacites à l'aide d'information externe. La cristallisation est la traduction du modèle mental d'un concept (connaissance tacite) par un développeur en artefact (connaissance explicite) tel qu'un diagramme de cas d'utilisation ou document d'architecture logicielle. La réalisation implique aussi la traduction de connaissances tacites en connaissances explicites, mais nécessite également un savoir-faire (*know-how*) technique, ce qui est relié à la production de code source. La validation implique un flux de connaissances bidirectionnel entre des connaissances tacites et des artefacts

de développement (connaissances explicites), de manière à la valider la cohérence de deux sources de connaissances. La vérification est similaire à la validation à l'exception que le code source est la source de connaissances, impliquant donc un savoir-faire (*know-how*) technique.

Le modèle de flux de connaissances est limité aux activités de développement logiciel. Les activités de gestion de projet ne sont pas prises en considération dans le modèle puisqu'elles ne sont pas spécifiques au développement logiciel. Rédiger un plan de développement logiciel est un exemple d'activité de gestion de projet.

Afin de faciliter la compréhension, le tableau 6.1 présente l'interprétation de mots-clefs de description d'un jeton ATS, en termes de facteur cognitif du modèle de flux de connaissances.

Tableau 6.1 : Mots-clefs des facteurs cognitifs

Facteurs cognitifs	**Mots-clefs de la description d'un jeton**
Acquisition	apprentissage, compréhension, lecture, réflexion, clarification
Cristallisation	élaboration, rédaction, ébauche, développement, modification, mise à jour (sauf code source)
Validation	révision, validation, correction (d'artefact) et discussion (synchronisation ou validation)
Réalisation	implémentation, codage, documentation (code), prototypage
Vérification	débogage, correction (de code), test (coder les tests, exécuter les tests), discussion (technique)

6.3 Caractéristiques des projets analysés

Cette section présente, aux tableaux 6.2 à 6.4, les caractéristiques principales des projets C6, C7 et C8. Ces informations faciliteront l'analyse des résultats.

Toutes les équipes comportent cinq membres.

Tableau 6.2 : Caractéristiques du projet C6

Caractéristique	Détails
Objectifs	• Offrir un environnement graphique de conception de systèmes avioniques. • Générer le code source et les fichiers reliés au modèle conçu.
Processus logiciel	Traditionnel, adapté du UPEDU
Système de saisie de jetons	Outil web TSCT, très structuré et simple à saisir, sans flexibilité de saisie
Équipe	• Utilisation formelle de la programmation par paire (2 développeurs). • Chef d'équipe apprécié et efficace. • Un membre est particulièrement porté sur la cristallisation. • Un membre peu impliqué dans le projet.

Tableau 6.3 : Caractéristiques du projet C7

Caractéristique	Détails
Objectifs	• Permettre de modifier les hiérarchies de groupes Doxygen de projets de documentation à partir d'une interface graphique conviviale.
Processus logiciel	Traditionnel, adapté du UPEDU
Système de saisie de jetons	Fichier Excel semi-structuré offrant une flexibilité de saisie.
Équipe	• Chef d'équipe exerçant un fort leadership. • Trois membres très motivés et consciencieux. • Un membre particulièrement porté sur la cristallisation. • Un membre moins impliqué dans le projet.

Tableau 6.4 : Caractéristiques du projet C8

Caractéristique	Détails
Objectifs	• Extraire les valeurs de champs alphanumériques et de champs symboliques à partir d'une vidéo présentant des écrans de tableaux de bord.
Processus logiciel	Traditionnel, adapté du UPEDU
Système de saisie de jetons	Fichier Excel semi-structuré offrant une flexibilité de saisie.
Équipe	• Membres aux personnalités très différentes. • Changement de chef d'équipe après la mi-projet. • Un membre particulièrement portée sur la cristallisation. • Un membre à la personnalité très discrète.

6.4 Caractérisation des développeurs

Il est possible de caractériser la rigueur des développeurs grâce à l'analyse de leurs jetons. Les identificateurs des développeurs ont été anonymisés par les lettres A, B, C, D, E, précédées de l'identificateur du projet (C6, C7 ou C8). Les tableaux 6.5 à 6.7 présentent les principales caractéristiques des jetons individuels des projets C6 à C8. Ainsi, pour chaque développeur, le nombre de jetons individuels, l'effort individuel, la médiane de la durée des jetons et la médiane du nombre de caractères de description des jetons sont spécifiés. La médiane de la durée des jetons individuels est un indicateur aussi appelé granularité de jetons, permettant de juger de la représentativité des jetons. Une médiane plus faible indique une plus grande représentativité des jetons d'un développeur. La médiane du nombre de caractères de description de jetons permet de juger de la précision d'un jeton. Dans ce cas, une médiane plus élevée indique une plus grande précision des jetons d'un développeur.

Tableau 6.5 : Caractéristiques des jetons individuels du projet C6

Développeur	Jetons	Effort (h)	Médiane durée jeton (h)	Médiane caractères
C6A	338	218	0,50	90
C6B	304	204	0,50	84
C6C	266	172	0,50	79
C6D	243	174	0,50	78
C6E	108	112	0,92	39

Tableau 6.6 : Caractéristiques des jetons individuels du projet C7

Développeur	Jetons	Effort (h)	Médiane durée jeton (h)	Médiane caractères
C7A	233	93	0,33	55
C7B	245	123	0,42	62
C7C	243	159	0,58	97
C7D	108	88	0,83	36
C7E	65	69	1,00	32

Tableau 6.7 : Caractéristiques des jetons individuels du projet C8

Développeur	Jetons	Effort (h)	Médiane durée jeton (h)	Médiane caractères
C8A	152	107	0,54	38
C8B	110	108	0,67	59
C8C	80	77	0,83	49
C8D	102	102	0,75	38
C8E	151	207	1,25	66

À la lumière de l'analyse des tableaux 6.5 à 6.7, il est possible de caractériser la rigueur relative des développeurs. Par exemple, C6A, avec une granularité de 0,5 heure et une précision de 90 caractères est plus rigoureux que C6E, avec une granularité de 0,92 heure et une précision de 39 caractères.

Il est possible de pousser plus loin l'analyse en étudiant la distribution de la durée des jetons individuels de chaque développeur. Les figures 6.2 à 6.4 présentent ces distributions pour les développeurs des projets C6, C7 et C8.

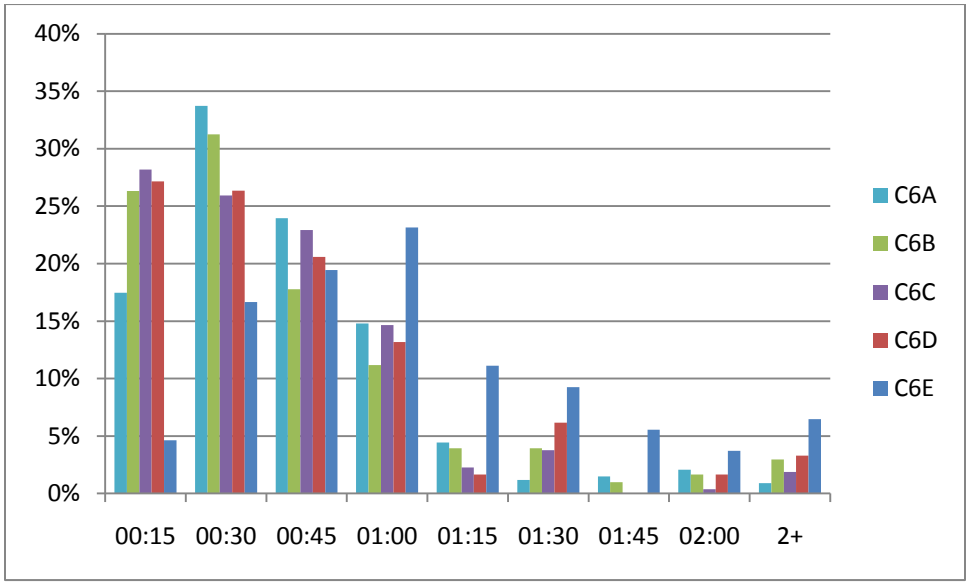

Figure 6.2 : Distribution de la durée des jetons individuels du projet C6

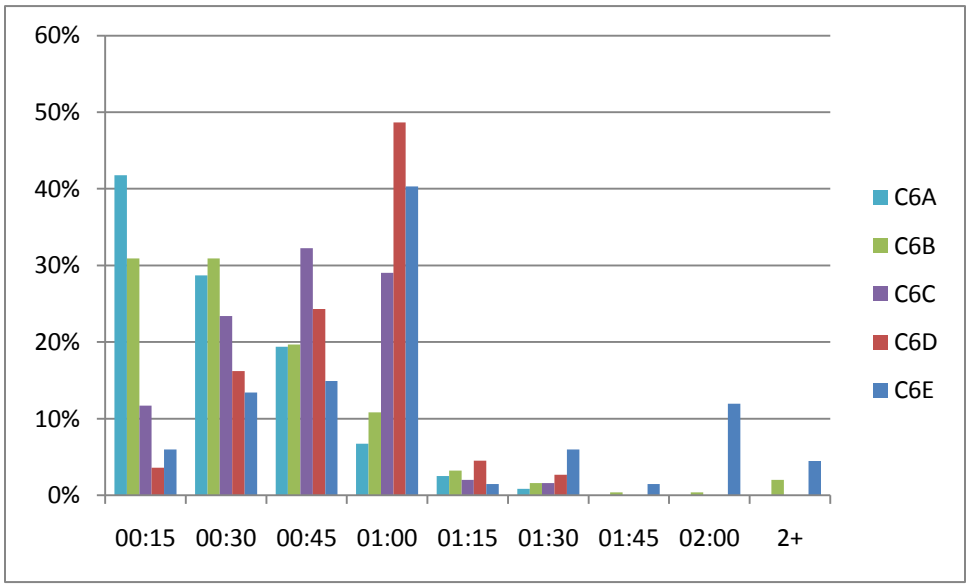

Figure 6.3: Distribution de la durée des jetons individuels du projet C7

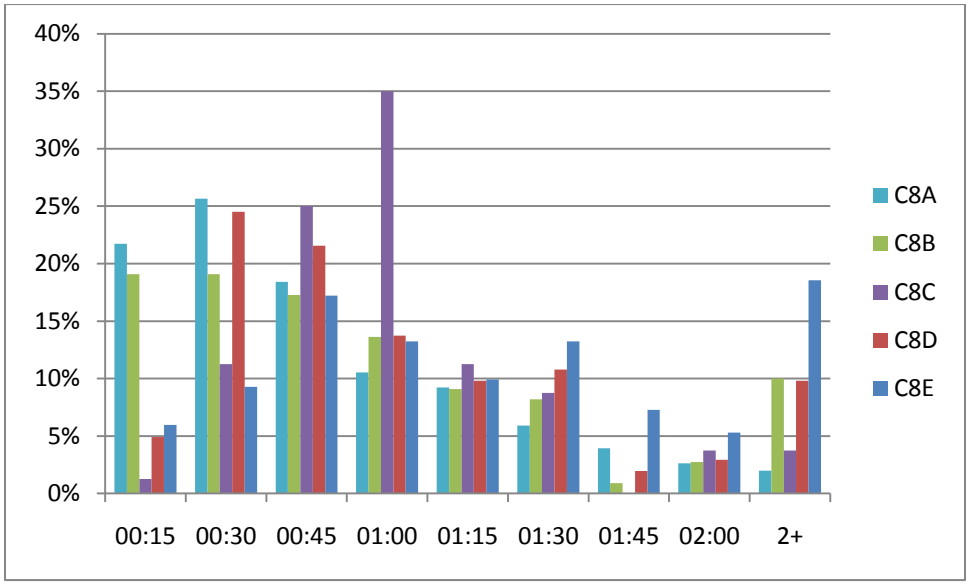

Figure 6.4 : Distribution de la durée des jetons individuels du projet C8

En observant les figures 6.2 à 6.4, on remarque qu'il existe principalement 3 types de profil de distribution de jetons. Ces profils sont présentés à la figure 6.5.

Les profils de distribution de jetons α, β et γ, présentés à la figure 6.5, offrent un outil d'analyse de la représentativité des jetons et de la rigueur des développeurs. Ainsi, les développeurs de profil α portent une attention prioritaire à la production de jetons, alors que les développeurs de profil β y portent une attention importante. Ces deux profils de développeurs produisent des jetons très représentatifs de leurs efforts, donc très fiables. Les développeurs de profil γ considèrent les jetons comme un mal nécessaire, ce qui ne veut pas dire que leurs jetons ne sont pas représentatifs de leur effort investi, mais ils sont moins fiables que ceux des deux autres profils. Grâce aux efforts de validation en cours de projet de la part de l'équipe de recherche, les jetons du "pire" profil sont suffisamment fiables pour les besoins de cette recherche.

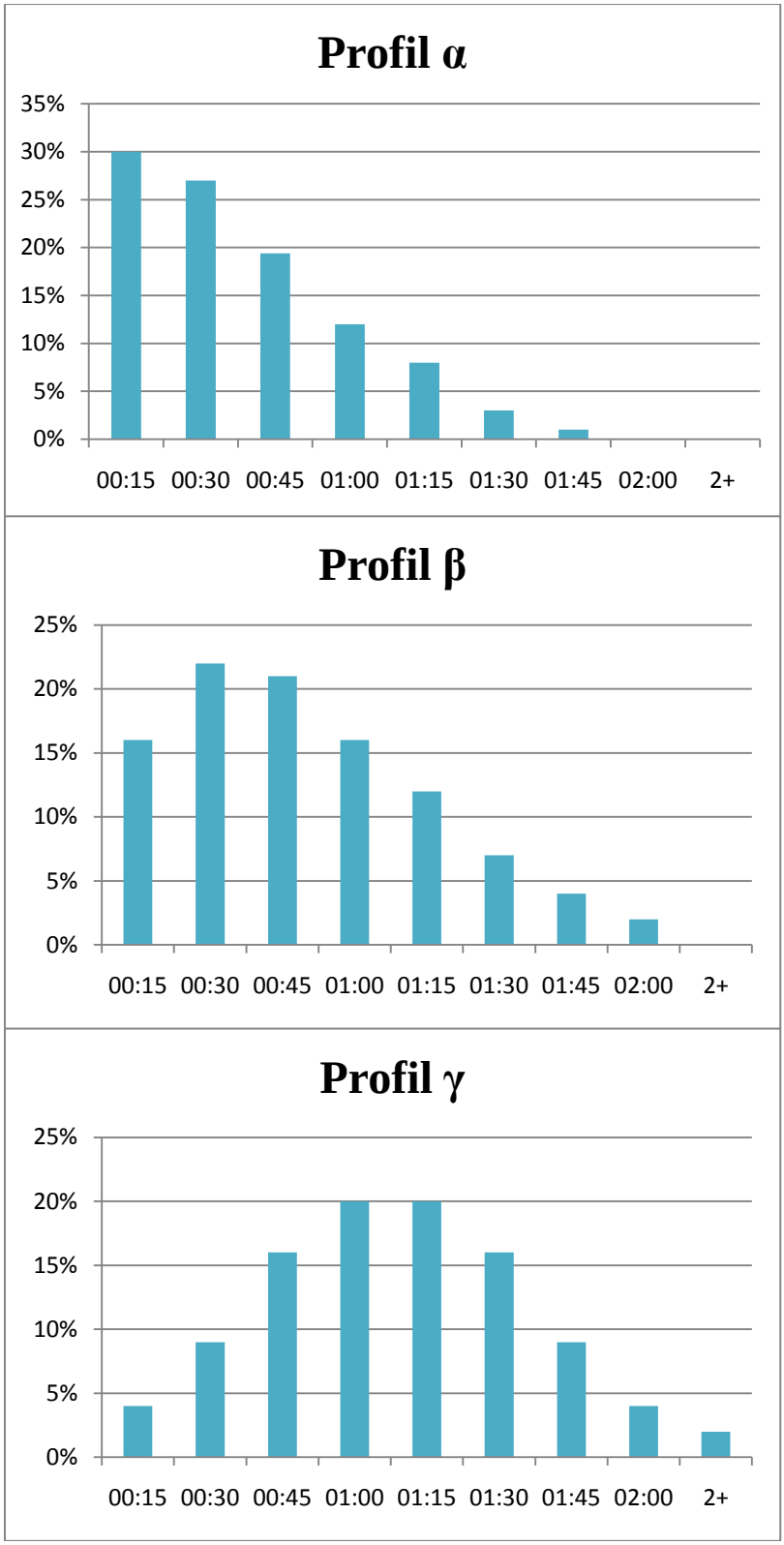

Figure 6.5 : Profils de distribution de jetons α, β et γ

Le tableau 6.8 résume les profils par équipe de développeur.

Tableau 6.8 : Caractéristiques des jetons individuels du projet C8

Projet	Développeurs au profil α	Développeurs au profil β	Développeurs au profil γ
C6	C6C, C6D	C6A, C6B	C6E
C7	C7A, C7B	C7C	C7D, C7E
C8	C8B	C8A, C8D	C8C, C8E

L'analyse du tableau 6.8, permet de confirmer une relative uniformité des jetons des 3 projets. Conséquemment, il raisonnable d'utiliser la totalité des jetons pour les analyses subséquentes.

6.5 Caractérisation de l'effort

Cette section vise à caractériser l'effort investi dans les projets C6, C7 et C8. Pour ce faire, la répartition de l'effort global sera analysée (section 6.5.1), la répartition et l'évolution du travail individuel et participatif seront étudiées (section 6.5.2), le séquencement cognitif sera expliqué (section 6.5.3) et la relation entre l'effort et le code source sera analysée (section 6.5.4) .

6.5.1 Effort global

Afin de parvenir à caractériser l'effort investi dans les projets C6, C7 et C8, il importe d'abord de comprendre la répartition d'effort global. Les tableaux 6.9 à 6.11 présentent l'effort investi, ainsi que la répartition relative des 6 facteurs cognitifs du modèle de flux de connaissances dans les projets C6, C7 et C8.

Tableau 6.9 : Effort investi par facteur cognitif pour le projet C6

Facteur cognitif	Effort (h)	Répartition
Acquisition	150	15%
Cristallisation	236	24%
Vérification	207	21%
Validation	141	14%
Réalisation	228	23%
Organisation du travail	36	4%
Total	**997**	**100%**

Tableau 6.10: Effort investi par facteur cognitif pour le projet C7

Facteur cognitif	Effort (h)	Répartition
Acquisition	60	8%
Cristallisation	192	26%
Vérification	163	22%
Validation	106	14%
Réalisation	143	19%
Organisation du travail	85	11%
Total	**750**	**100%**

Tableau 6.11 : Effort investi par facteur cognitif pour le projet C8

Facteur cognitif	Effort (h)	Répartition
Acquisition	100	11%
Cristallisation	186	21%
Vérification	209	24%
Validation	104	12%
Réalisation	180	20%
Organisation du travail	108	12%
Total	**887**	**100%**

Les tableaux 6.9 à 6.11 permettent d'observer plusieurs tendances concernant la répartition d'effort dans un projet intégrateur.

Pour les projets C6, C7 et C8, le rapport entre la cristallisation et la validation est relativement constant soit de 1,7 à 1,9. De plus, le rapport entre la réalisation et la vérification est également relativement constant soit de 0,8 à 1,1. Les efforts de vérification et validation sont virtuellement constants à respectivement 35%, 36% et 36%. Finalement, les efforts de cristallisation et réalisation sont relativement constants à respectivement 47%, 47% et 41%.

L'acquisition est variable d'un projet à l'autre passant de 8% à 15%. En fait, l'acquisition nécessaire est variable selon la différence entre les connaissances déjà acquises par les développeurs d'une équipe avant le début du projet et les connaissances qui seront nécessaires dans le cadre du projet.

L'organisation du travail est aussi variable passant de 4% à 12%. Ce facteur cognitif dépend de plusieurs caractéristiques relatives au fonctionnement de l'équipe de développeurs dont le degré de maillage d'une équipe et le type de leadership exercé.

En somme, malgré les différences des 3 projets (nature du projet, composition de l'équipe, outils utilisés), il existe plusieurs constantes relativement à l'effort investi dans les différents facteurs cognitifs. Ce phénomène laisse présager la possibilité d'établir des modèles prédictifs.

6.5.2 Travail individuel et participatif

Au sein du développement d'un projet logiciel, les activités sont exécutées de manière individuelle (1 développeur) ou participative (2 développeurs ou plus). L'analyse de la répartition et de l'évolution de l'effort permet de mieux comprendre les besoins en travail participatif au sein d'un projet de développement logiciel.

Le tableau 6.12 présente la répartition de l'effort individuel par rapport à l'effort participatif à 2 développeurs et à 3 développeurs ou plus.

Tableau 6.12 : Répartition de l'effort individuel et participatif des projets C6 à C8

Projet	Effort individuel (%)	Effort participatif à 2 développeurs (%)	Effort participatif à 3 développeurs ou plus (%)
C6	60	26	14
C7	67	11	22
C8	69	3	28

On remarque que l'effort individuel totalise environ les deux tiers de l'effort total pour les projets C7 et C8, avec respectivement 67% et 69% d'effort individuel. En ce qui a trait au projet C6, l'effort individuel moins important (60%) s'explique par le fait que 2 des 5 développeurs ont pratiqué la programmation par paire. Ainsi, pour le projet C6, l'effort participatif à 2 développeurs dépasse le quart (26%) d'effort total, contrairement aux projets C7 et C8 (respectivement 11% et 3%).

Les figures 6.6 à 6.8 présentent l'évolution de l’effort individuel, participatif à 2 développeurs et participatif à 3 développeurs et plus des projets C6 à C8.

Le type de représentation graphique choisi nécessite d'abord la division d'un projet en tranches d’effort de 5%. Par exemple, pour un projet de 800 heures, les heures 0 à 40 correspondraient à la mesure d'avancement du projet (axe des abscisses) à 5%, les heures 40 à 80 correspondraient à la mesure à 10% et ainsi de suite. Pour chaque tranche de 5% d'avancement, les proportions

relatives à l'effort individuel, participatif à 2 développeurs et participatif à 3 développeurs et plus sont déterminées. Pour une mesure d'avancement de 5% (en abscisses), la somme des efforts totaux (effort à 1, à 2 ou à 3+ développeurs) totalise toujours 5% (en ordonnées). En d'autres mots, l'effort total (axe des ordonnées) est représenté de manière non cumulative.

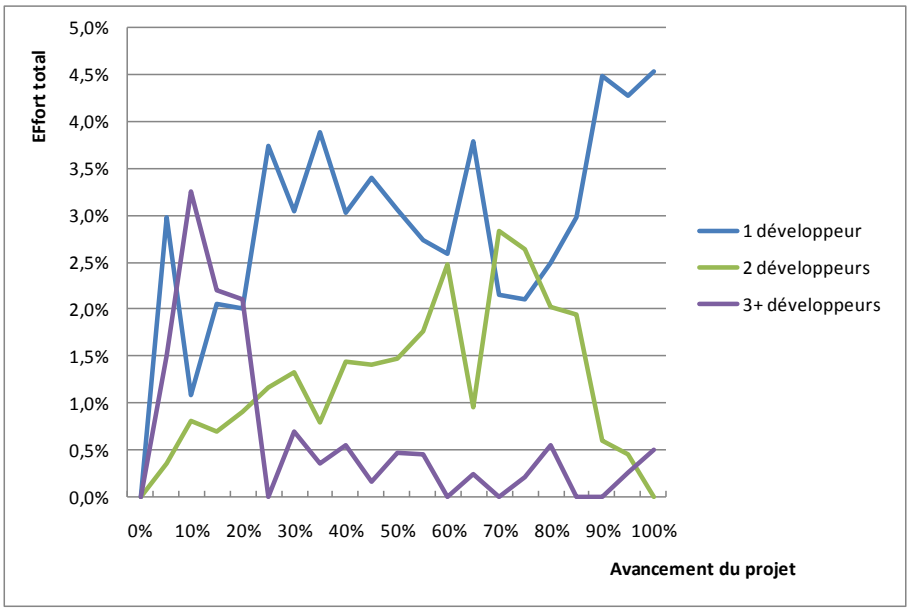

Figure 6.6 : Évolution de l'effort individuel et participatif du projet C6

À titre d'exemple d'interprétation de la figure 6.6, on remarque qu'à 65% d'avancement du projet (correspondant à la tranche d'effort de 60% à 65%), 3,8 % de l'effort total a été effectué par 1 participant, 1,0% par 2 participants et 0,2% par 3 participants ou plus.

Les figures 6.6 à 6.8 nous permettent de dresser deux constats. D'une part, l'effort individuel est l'effort majoritaire tout au long du projet, sauf en ce qui a trait aux premiers 20% (C6 et C7) à 40% (C8) d'avancement du projet. On remarque cependant une exception de 70% à 80% d'avancement du projet C6, ce qui s'explique par la programmation par paire pratiquée uniquement par cette équipe. D'autre part, l'effort participatif est surtout concentré dans les premiers 25% (C6) à 45% (C8) d'avancement du projet. Cela s'explique par la nécessité pour les différents développeurs d'une équipe de synchroniser leur modèle mental du produit à réaliser. Cette phase de synchronisation est surtout associée aux disciplines de requis et de conception. La stabilisation du modèle mental de l'équipe facilite la discipline d'implémentation. Pour le reste du projet, une synchronisation est nécessaire de manière sporadique, comme on peut le voir dans les trois précédentes figures.

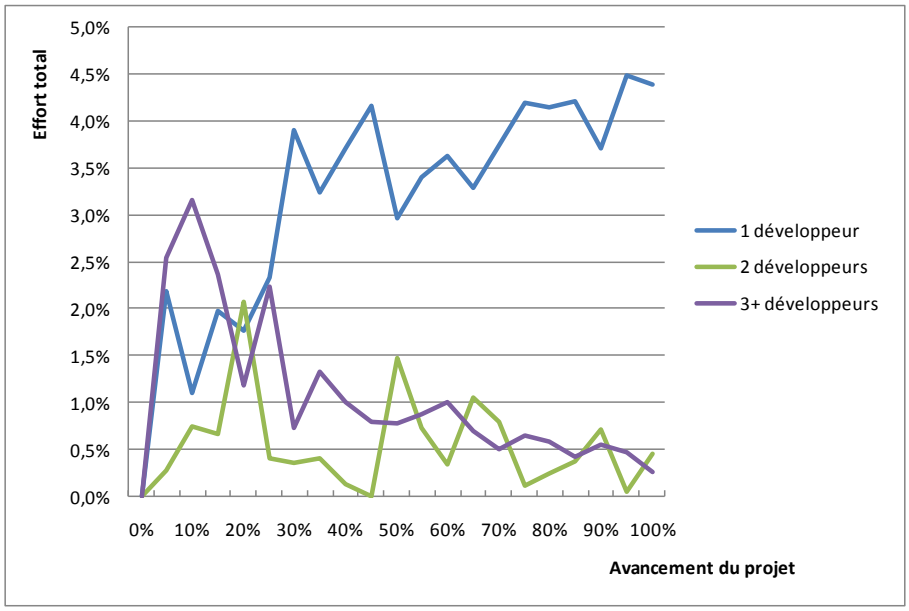

Figure 6.7 : Évolution de l'effort individuel et participatif du projet C7

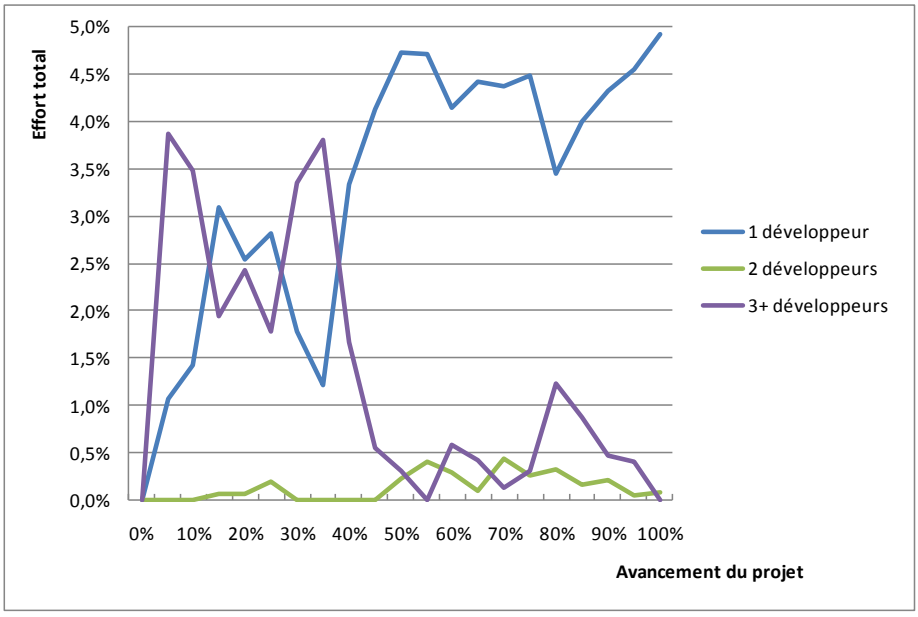

Figure 6.8 : Évolution de l'effort individuel et participatif du projet C8

Après s'être intéressé à l'évolution de l'effort individuel et participatif de manière globale, il est intéressant de décortiquer l'évolution de l'effort pour chacun des facteurs cognitifs.

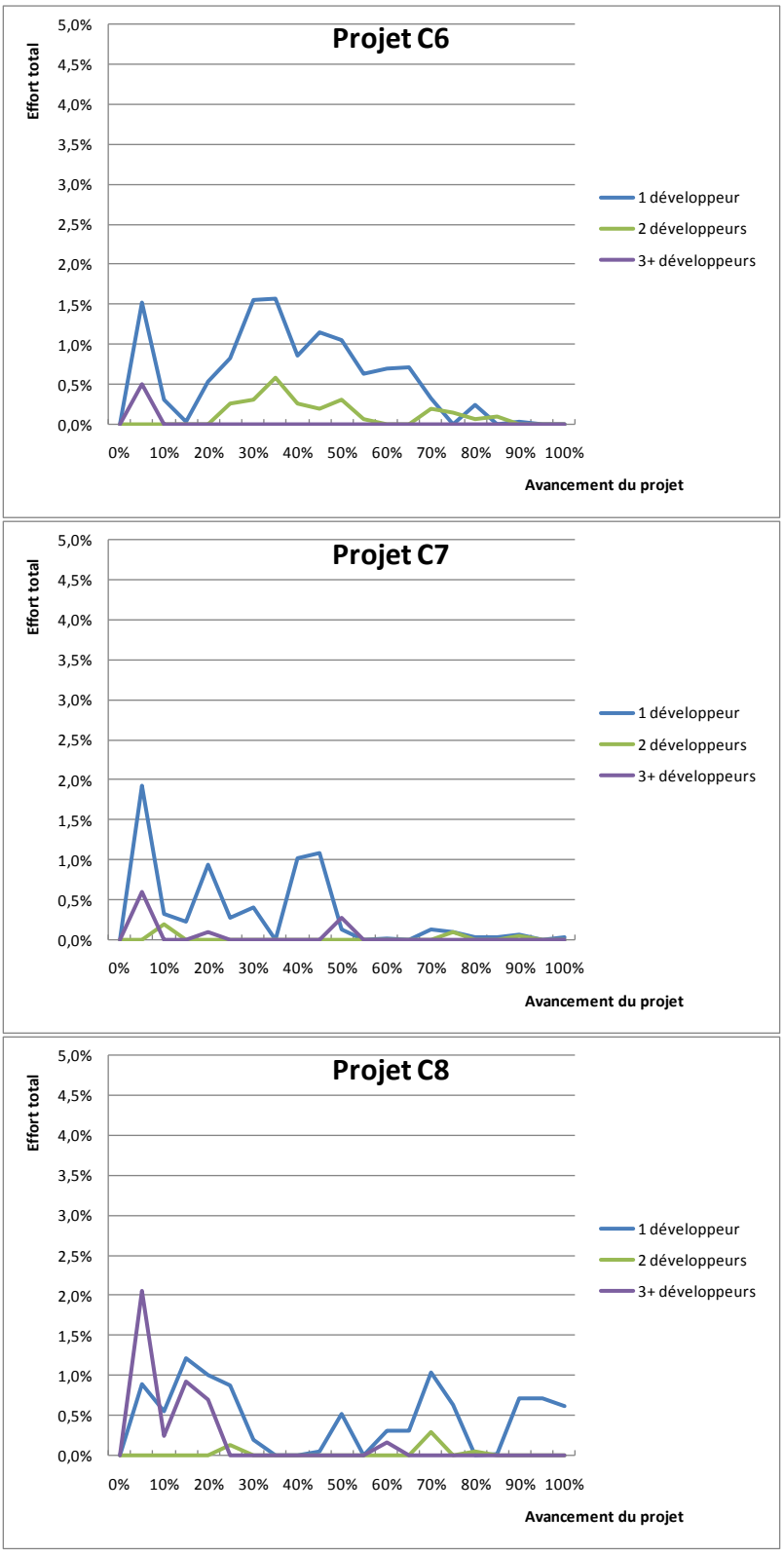

Figure 6.9 : Évolution de l'effort d'acquisition individuel et participatif des projets C6 à C8

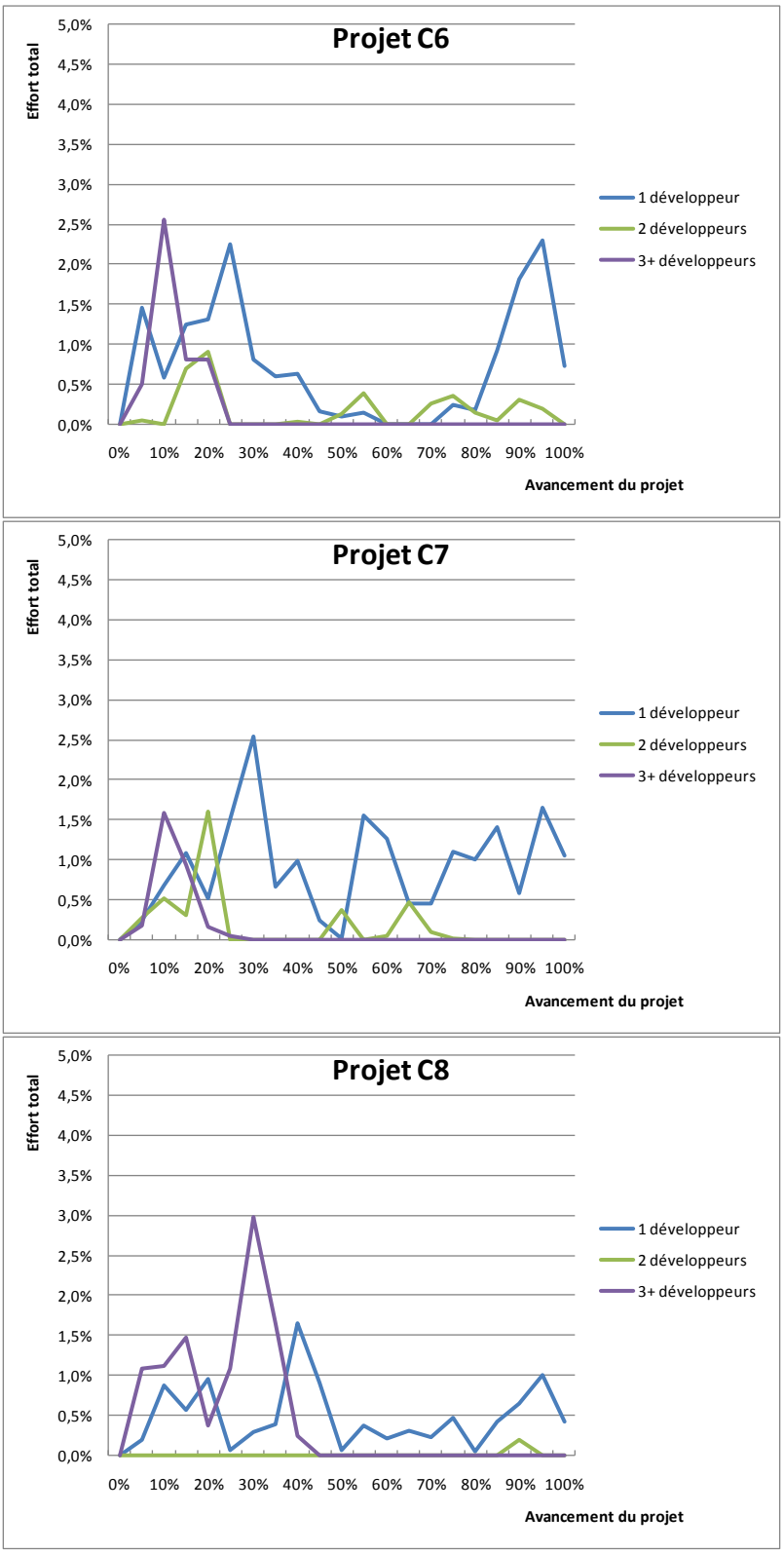

Figure 6.10 : Évolution de l'effort de cristallisation individuel et participatif des projets C6 à C8

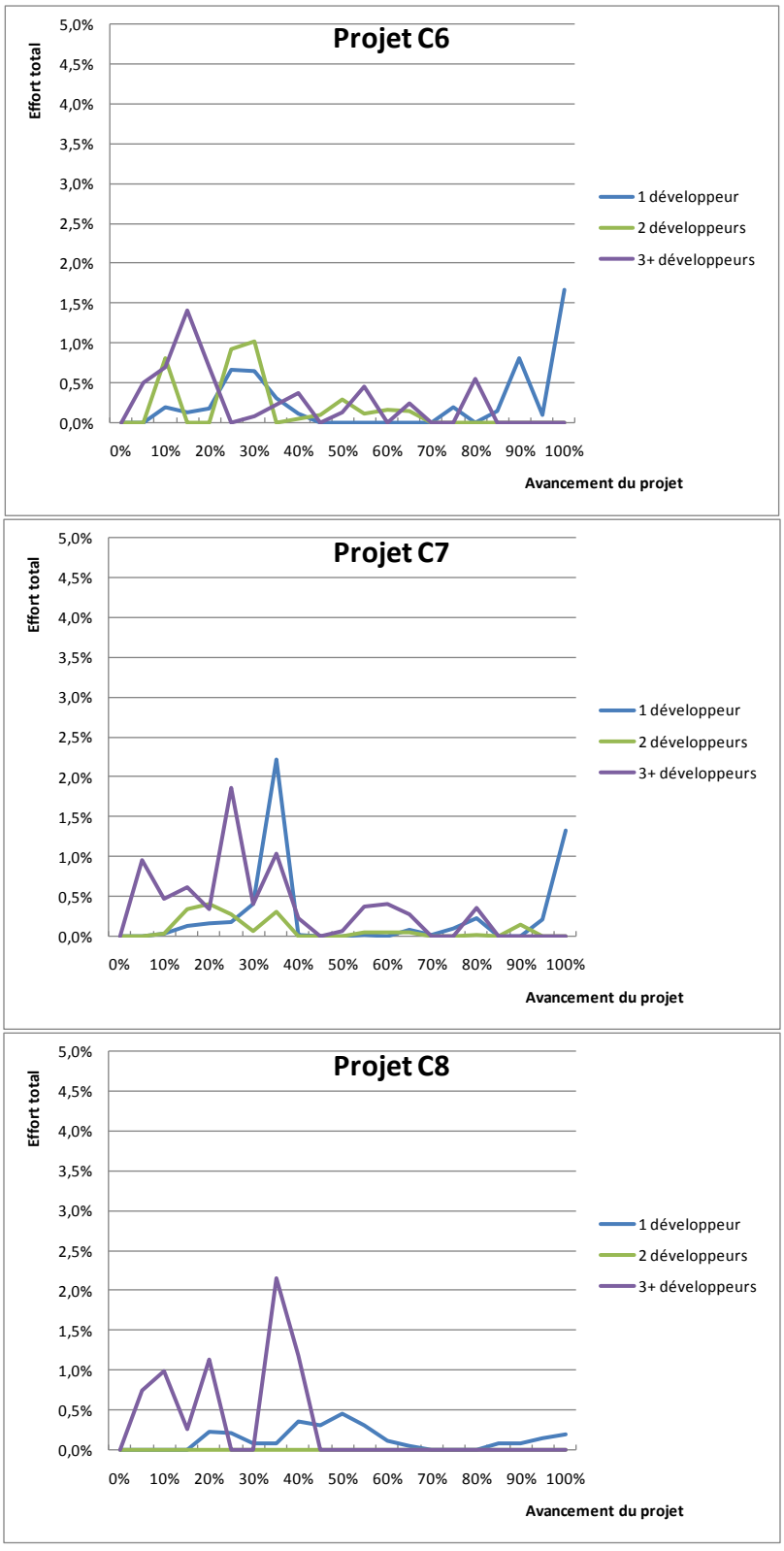

Figure 6.11: Évolution de l'effort de validation individuel et participatif des projets C6 à C8

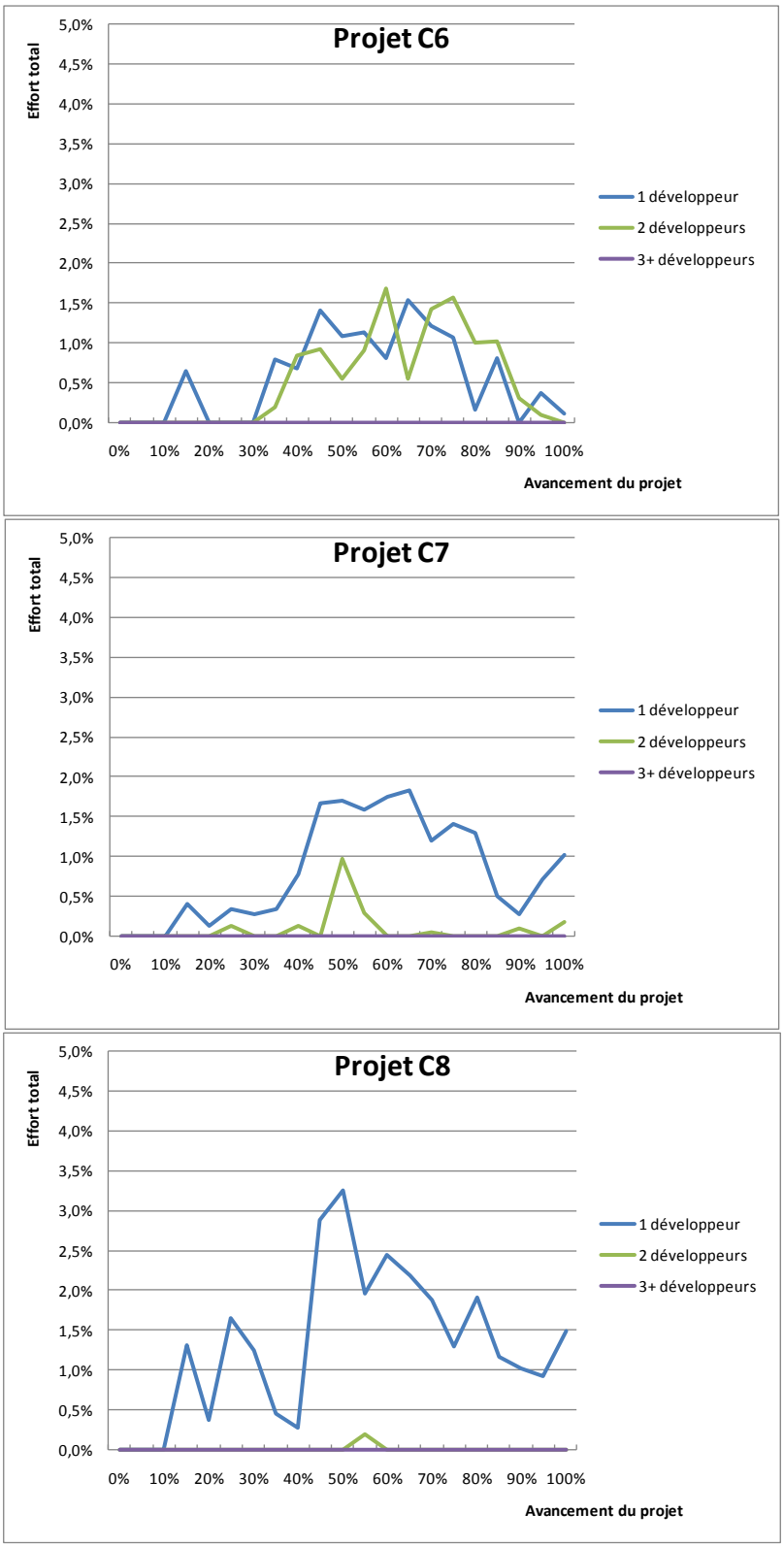

Figure 6.12 : Évolution de l'effort de réalisation individuel et participatif des projets C6 à C8

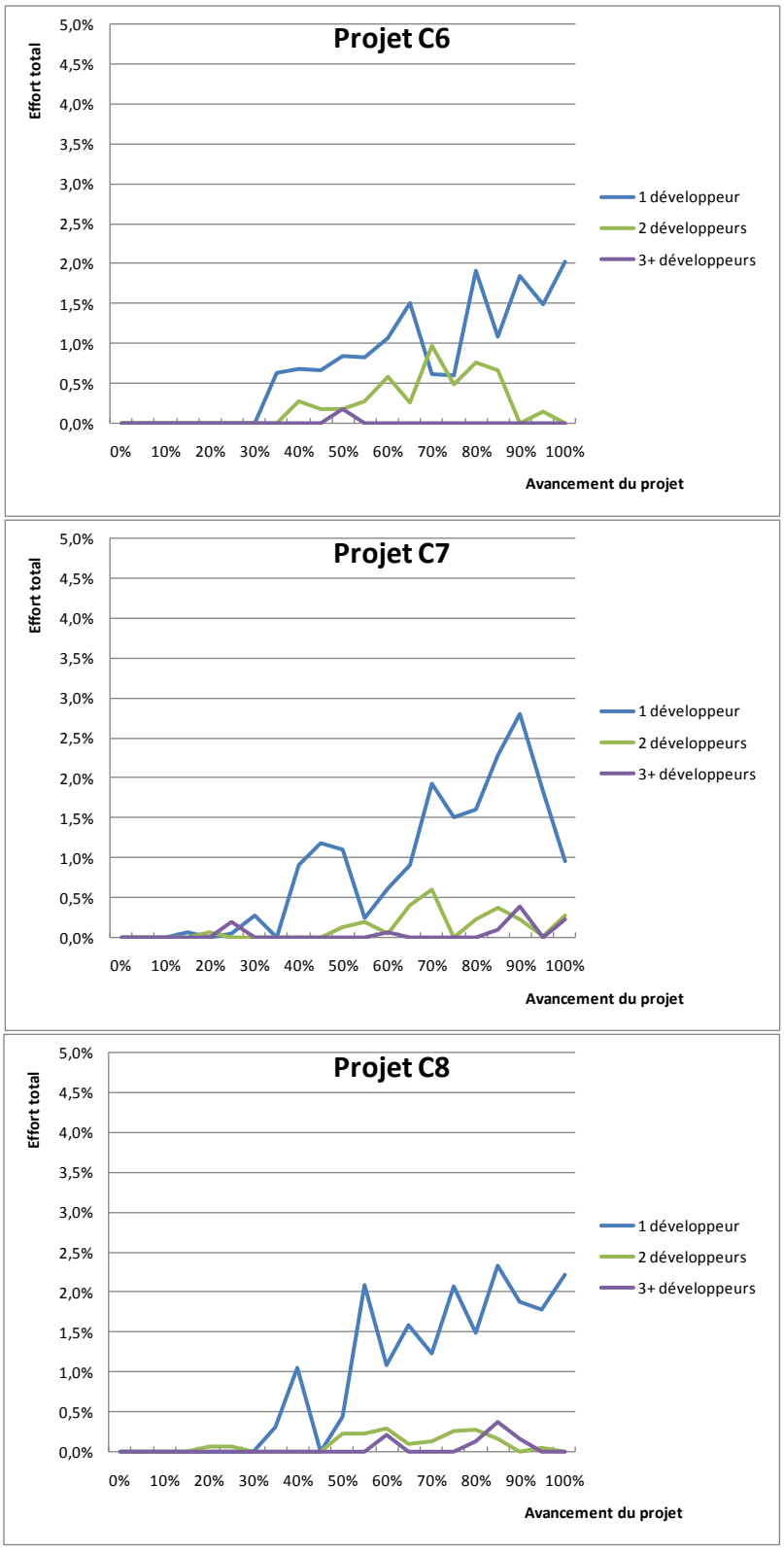

Figure 6.13 : Évolution de l'effort de vérification individuel et participatif des projets C6 à C8

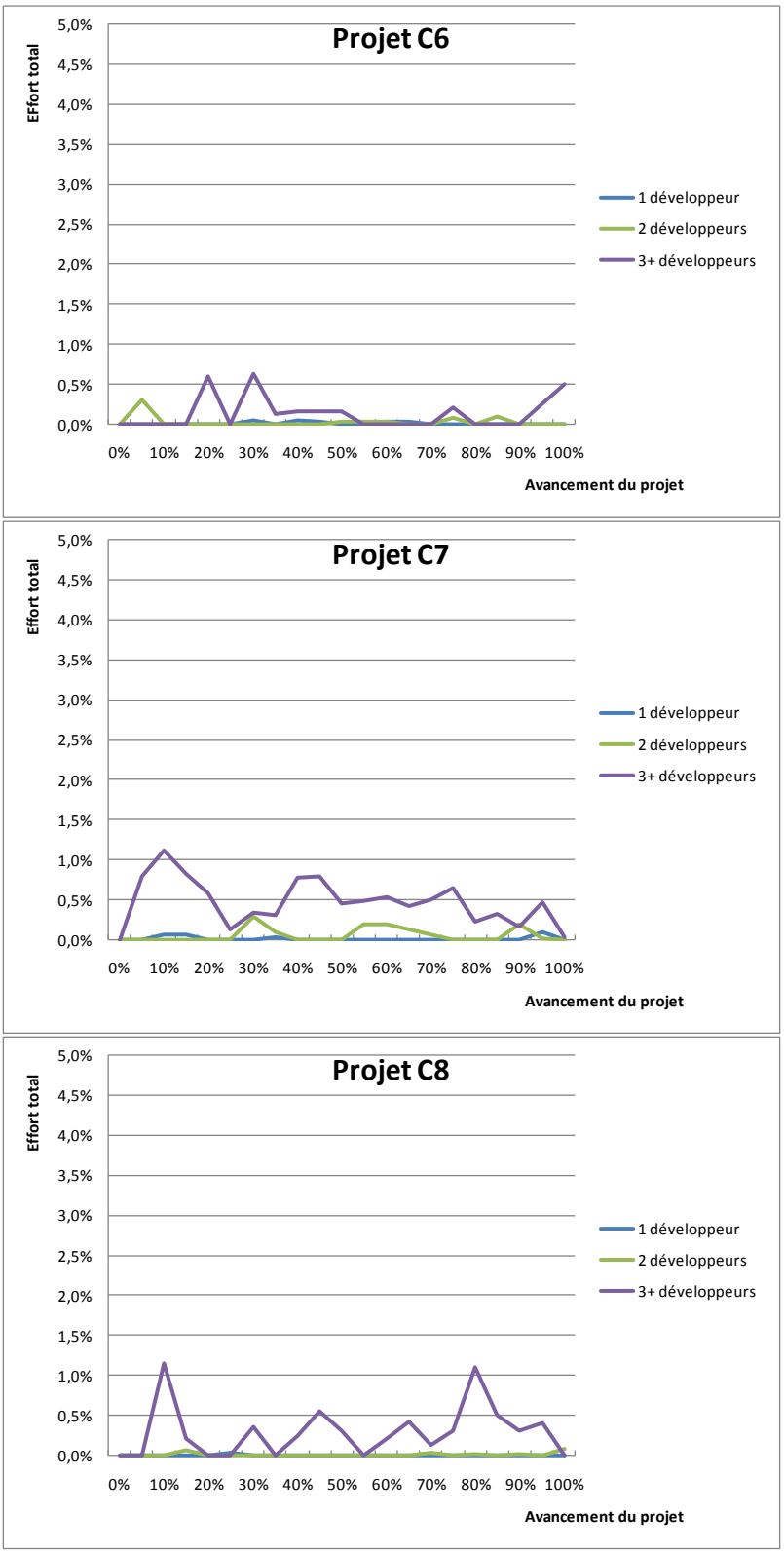

Figure 6.14 : Évolution de l'effort d'organisation du travail individuel et participatif des projets C6 à C8

La figure 6.9 présente l'évolution de l'effort d'acquisition du travail individuel et participatif des projets C6 à C8. D'une part, sans surprise, l'acquisition est très majoritairement individuelle. La seule portion notable d'effort participatif a lieu au tout début du projet (premiers 10% d'avancement), ce qui s'explique par la première rencontre de l'équipe avec le client, où celui-ci détaille ses besoins aux développeurs. D'autre part, on remarque que la plus grande partie de l'effort d'acquisition est investi dans la première moitié du projet. De plus, dépendamment de la nature du projet, donc des besoins en connaissances des développeurs d'un projet, l'effort d'acquisition évolue de manière opportuniste, dans une perspective juste-à-temps, ce qui confirme le comportement observé par Robillard (2005).

La figure 6.10 présente l'évolution de l'effort de cristallisation du travail individuel et participatif des projets C6 à C8. On constate que la cristallisation comporte principalement deux phases. La première phase s'échelonne du début du projet et s'étend environ jusqu'à la mi-projet. Il s'agit de la phase où les développeurs cristallisent "l'image de possibilité" du logiciel en devenir (cf. chapitre 5). L'effort, au cours de cette phase, est à la fois autant participatif qu'individuel. La deuxième phase occupe les derniers 20% du projet. À ce stade, les développeurs procèdent au *retrofitting* des artefacts, c'est-à-dire qu'ils mettent à jour les artefacts pour qu'ils reflètent l'implémentation réelle du produit. Contrairement à la première phase, l'effort est très majoritairement individuel, puisqu'il n'y a que très peu de synchronisation à faire ce stade.

La figure 6.11 présente l'évolution de l'effort de validation du travail individuel et participatif des projets C6 à C8. La validation est intimement liée à la cristallisation, ce qui explique plusieurs similitudes entre les évolutions respectives. En effet, au sein du développement logiciel, le facteur cognitif de validation regroupe les activités visant à assurer que les artefacts et les modèles mentaux de l'équipe sont valides. Par ailleurs, la validation est majoritairement participative et elle est concentrée dans la première moitié du projet, tout comme la cristallisation. De plus, une pointe d'effort à la fin du projet coïncide avec le *retrofitting* de fin de projet. Dans le cas du projet C8, l'absence de pointe à la toute fin du projet indique une validation déficiente.

La figure 6.12 présente l'évolution de l'effort de réalisation du travail individuel et participatif des projets C6 à C8. La réalisation consiste tout simplement en la production du code source. La forte majorité de l'effort de réalisation se concentre entre 40% et 90% d'avancement du projet.

L'effort antérieur à 40% correspond à l'élaboration de prototypes. La réalisation est très majoritairement individuelle, sauf pour le projet C6, dans lequel deux développeurs programmaient en binôme.

La figure 6.13 présente l'évolution de l'effort de vérification du travail individuel et participatif des projets C6 à C8. De la même manière que la validation est liée à la cristallisation, la vérification est intimement liée à la réalisation. En effet, le facteur cognitif de vérification fait principalement référence au débogage, au codage de tests et à l'exécution des tests. La vérification est fortement individuelle, à l'exception de l'équipe ayant pratiqué la programmation par paire (C6).

La figure 6.14 présente l'évolution de l'effort d'organisation du travail individuel et participatif des projets C6 à C8. L'organisation du travail est virtuellement uniquement participative (3 développeurs et plus) et est globalement de moindre importance comparativement aux 5 autres facteurs cognitifs. Par ailleurs, l'effort est plus important lors des premiers 30% d'avancement, ce qui est dû au besoin plus important de s'organiser en début de projet.

6.5.3 Séquencement cognitif

Le séquencement cognitif permet d'analyser l'effort investi par un développeur dans les différents facteurs cognitifs, en tenant compte de leur temporalité relative. À titre d'exemple, les figures 6.15 et 6.16 présentent deux vues complémentaires du séquencement cognitif du développeur C7A. Ainsi, la vue A-CV-RV (figure 6.15) associe les valeurs suivantes sur l'axe des ordonnées: 1 pour l'acquisition, 2 pour la cristallisation et la validation et 3 pour la réalisation et la vérification. Par ailleurs, la vue A-CR-VV (figure 6.16) associe les valeurs suivantes sur l'axe des ordonnées: 1 pour l'acquisition, 2 pour la cristallisation et la réalisation et 3 pour la validation et la vérification. Le choix de présenter deux vues complémentaires est d'ordre conceptuel. La vue A-CV-RV regroupe les facteurs cognitifs relatifs à la production d'artefacts (cristallisation et validation) ainsi que de code source (réalisation et vérification), alors que la vue A-CR-VV regroupe les facteurs cognitifs d'externalisation (cristallisation et réalisation) ainsi que de combinaison (vérification et validation).

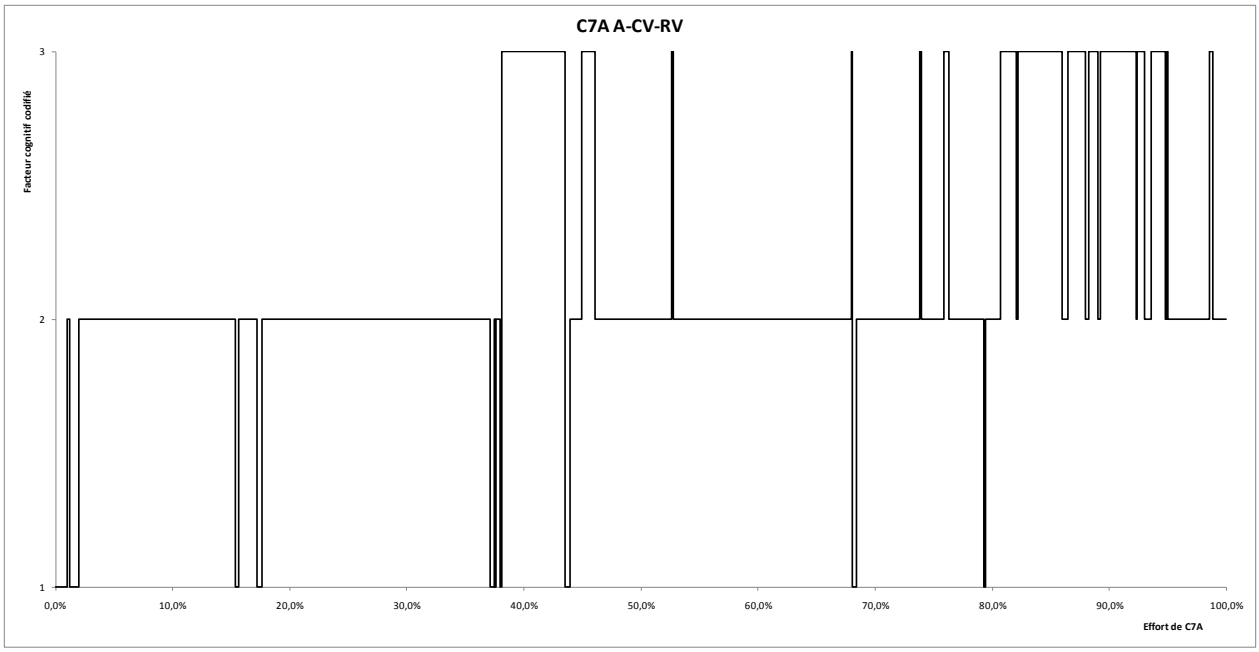

Figure 6.15 : Vue A-CV-RV du séquencement cognitif du développeur C7A

Figure 6.16 : Vue A-CR-VV du séquencement cognitif du développeur C7A

Afin de faciliter la compréhension des graphiques de séquencement cognitif, les figures 6.17 et 6.18 présentent un agrandissement des premiers 10% d'effort des figures 6.15 et 6.16.

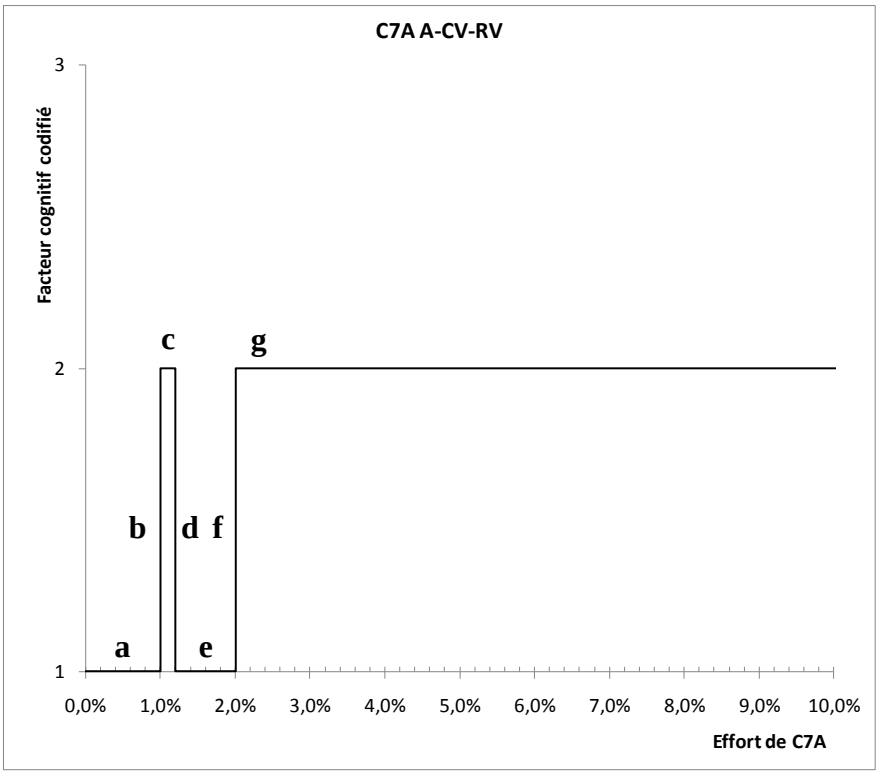

Figure 6.17 : Vue partielle A-CV-RV du séquencement cognitif du développeur C7A

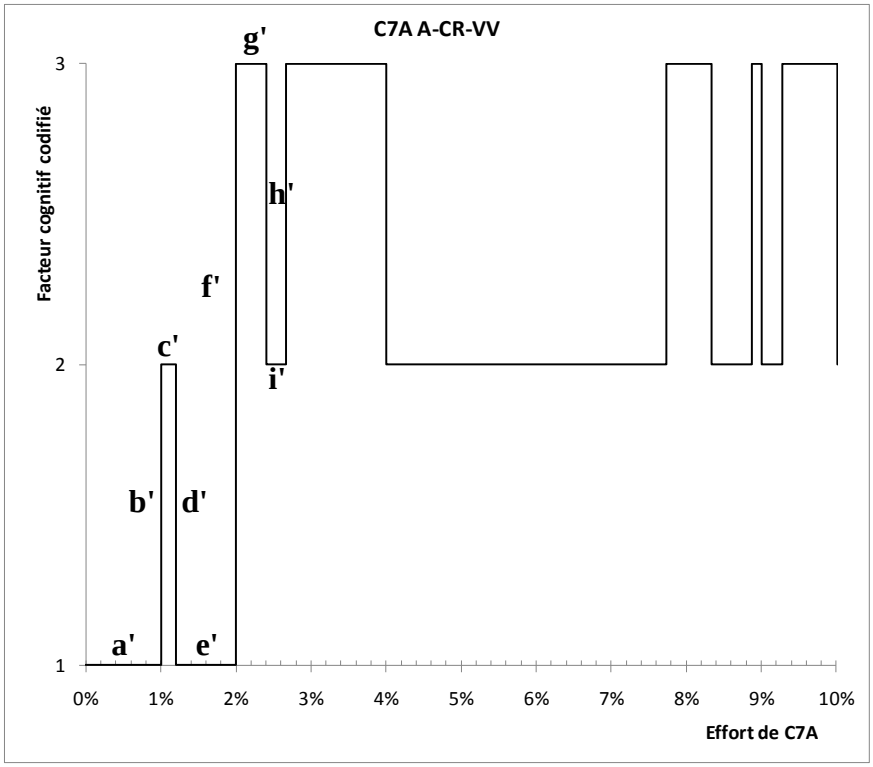

Figure 6.18 : Vue partielle A-CR-VV du séquencement cognitif du développeur C7A

Pour permettre l'analyse du séquencement cognitif, il est nécessaire d'observer les deux vues complémentaires A-CV-RV et A-CR-VV. Par exemple, le segment **a** (figure 6.17) et son complément **a'** (figure 6.18) indiquent que le développeur C7A a consacré son premier pourcent d'effort en acquisition. Le segment **c** (figure 6.17) indique qu'entre 1,0% et 1,2% de l'effort total de C7A, le développeur était en cristallisation ou en validation. Le segment **c'** (figure 6.18) indique qu'entre 1,0% et 1,2% de l'effort total de C7A, le développeur était en cristallisation ou en réalisation. Par l'analyse complémentaire de **c** et **c'**, on peut conclure qu'entre 1,0% et 1,2% de l'effort total de C7A, le développeur était en cristallisation. Ainsi, il est aussi possible de déduire qu'entre 1,2% et 2,0% (**e** et **e'**), C7A était en acquisition, qu'entre 2,0% et 2,4% (**g** et **g'**), C7A était en validation et qu'entre 2,4% et 2,7% (**g** et **i'**), C7A était en cristallisation. Par ailleurs, les segments **b**, **b'**, **d**, **d'**, **f**, **f' et h'** illustrent tous une transition entre deux facteurs cognitifs.

Les vues complémentaires A-CV-RV et A-CR-VV présentent donc une "signature" du séquencement cognitif d'un développeur, permettant d'analyser son effort investi dans les différents facteurs cognitifs, en tenant compte de leur temporalité relative.

6.5.3.1 Profils de développeurs

À partir de la signature du séquencement cognitif des développeurs, il est possible de définir des profils de développeurs.

Tableau 6.13 : Profils des développeurs

Profil	Caractéristiques
Cristallisateur	Priorité accordée à la cristallisation et à la validation des artefacts
Codeur	Priorité accordée à réalisation et à la vérification du code source
Polyvalent	Priorité varie selon les besoins du projet
Agent libre	Détachement par rapport au projet, comportement cognitif erratique

Le tableau 6.13 présente les 4 profils de développeurs identifiés dans les projets intégrateurs C6, C7 et C8.

Le cristallisateur a d'abord et avant tout une préoccupation pour la production d'artefacts, soit la cristallisation et la validation des artefacts. Un exemple de ce profil est le développeur C7A (figures 6.15 et 6.16). On remarque que tout au long du projet, le développeur investit la majorité de son temps en cristallisation ou en validation.

Le codeur investit globalement un effort très important en production de code source, soit la réalisation et la vérification. Un exemple de ce profil est le développeur C8E (figures 6.19 et 6.20). On remarque que le codeur C8E commence la réalisation aussi tôt qu'à 9% (il s'agit en fait de prototypage), comparativement au cristallisateur C7A qui commence la réalisation à 38%. Globalement, C8E investit peu d'efforts en production d'artefacts, consacre la majorité de ses efforts en production de code source, tout en acquérant des connaissances de manière opportuniste.

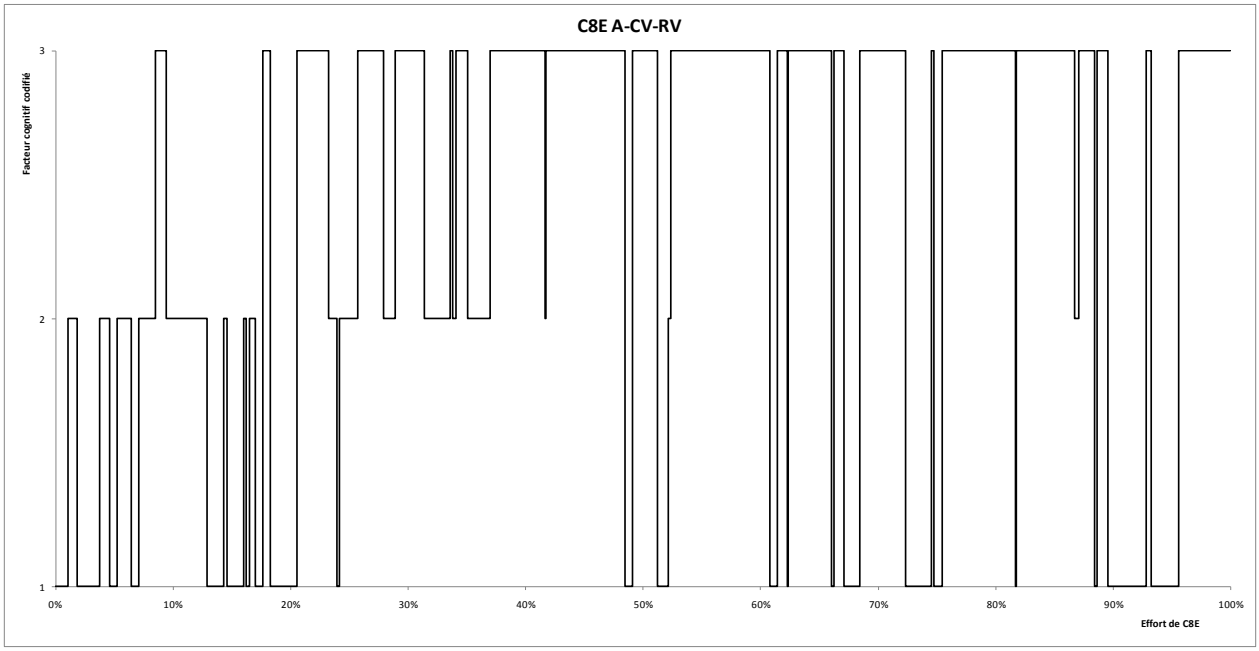

Figure 6.19 : Vue A-CV-RV du séquencement cognitif du développeur C8E

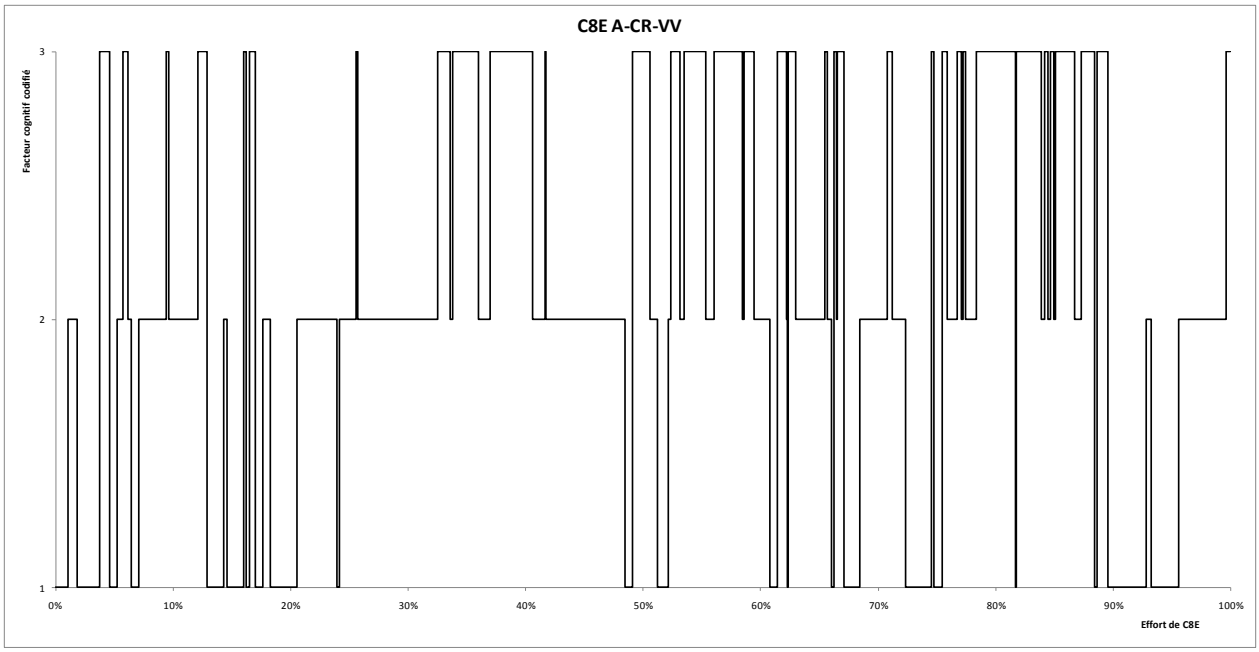

Figure 6.20 : Vue A-CR-VV du séquencement cognitif du développeur C8E

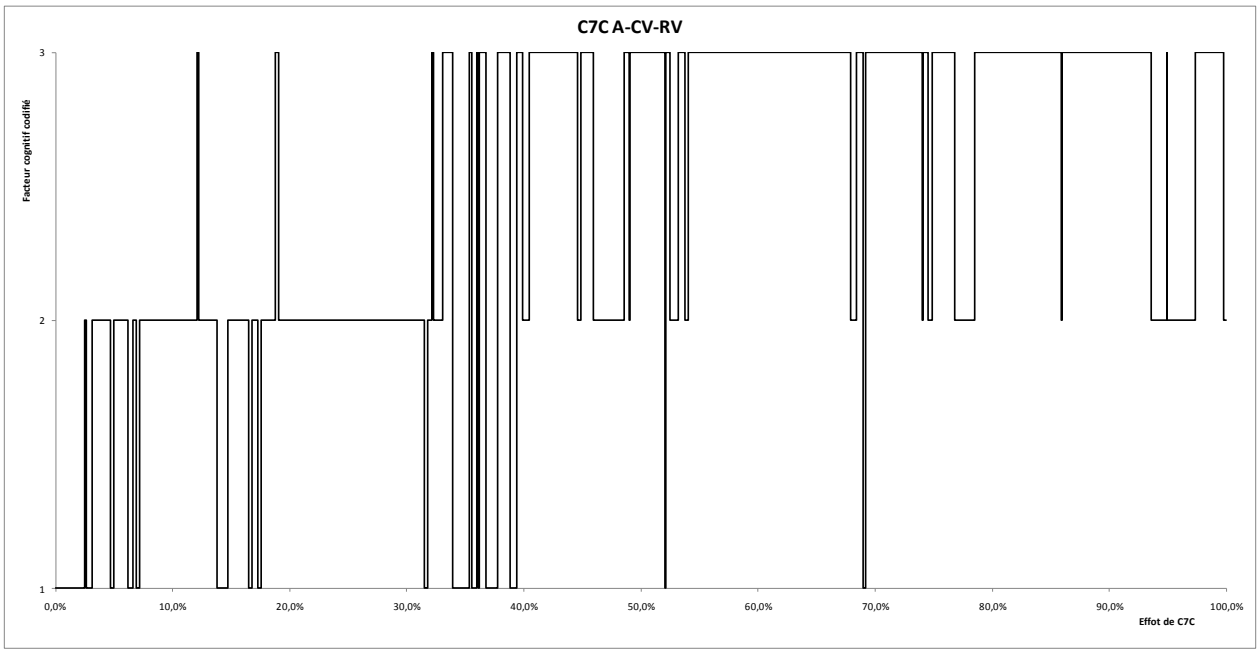

Figure 6.21 : Vue A-CV-RV du séquencement cognitif du développeur C7C

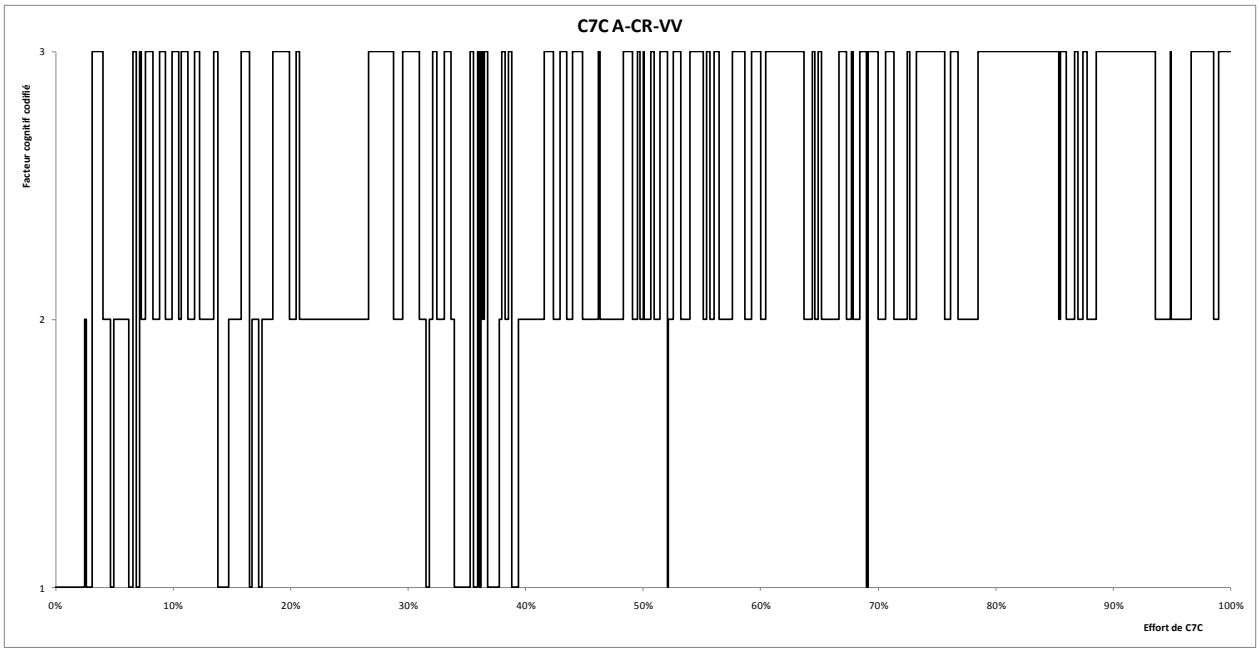

Figure 6.22 : Vue A-CR-VV du séquencement cognitif du développeur C7C

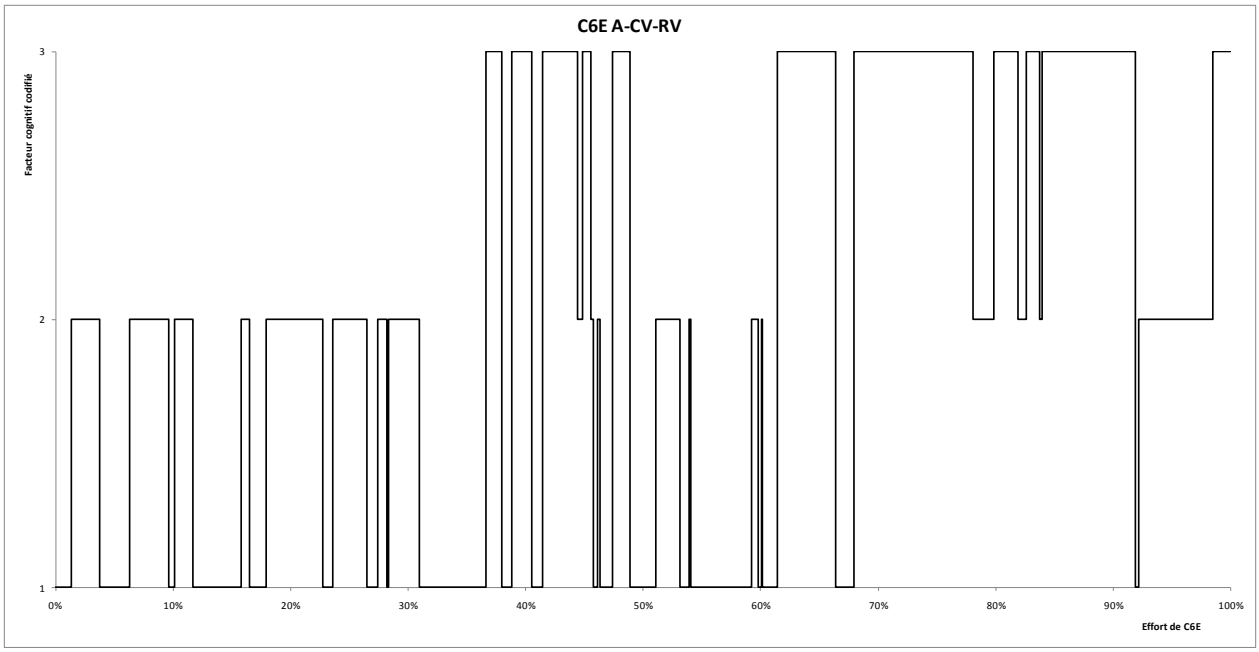

Figure 6.23 : Vue A-CV-RV du séquencement cognitif du développeur C6E

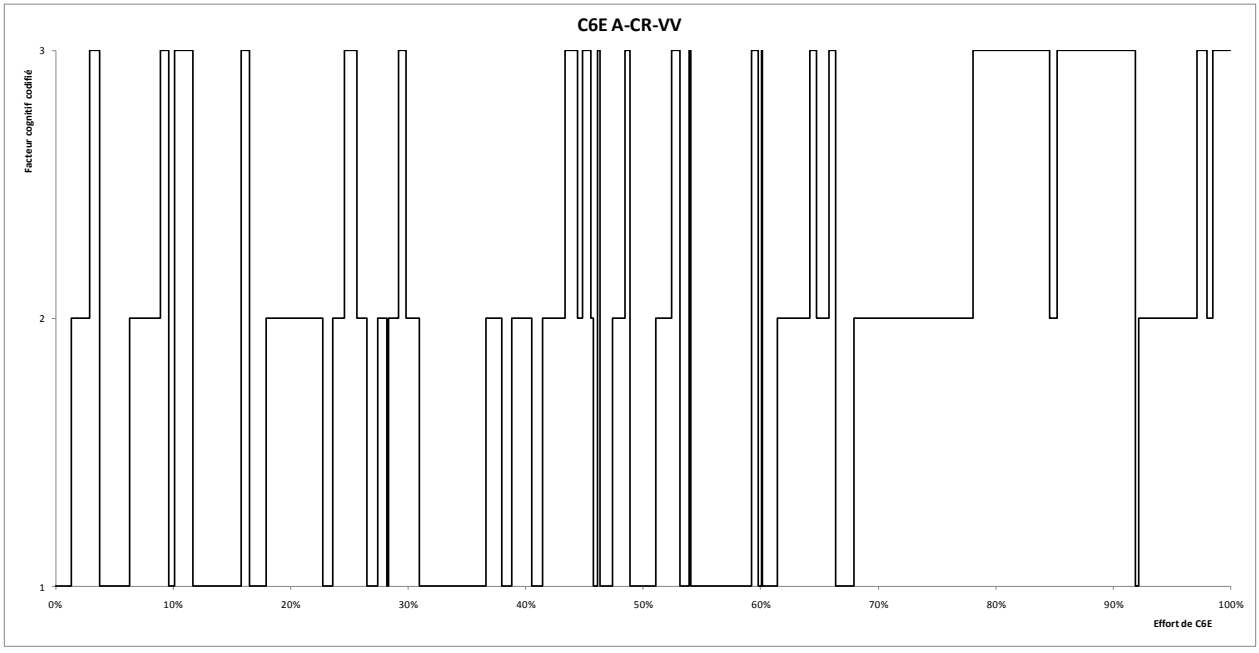

Figure 6.24 : Vue A-CR-VV du séquencement cognitif du développeur C6E

Le polyvalent investit ses efforts selon les besoins du projet. Un exemple de ce profil est le développeur C7C (figures 6.21 et 6.22). On remarque que le polyvalent C7C investi la grande majorité de ses efforts en production d'artefacts pour les premiers 30% d'effort. Pour le reste du projet, le polyvalent investit la majorité de ses efforts en production de code source, tout en investissant un effort non négligeable en production d'artefacts.

L'agent libre ressent peu d'attachement pour le projet, ce qui a pour conséquences un comportement cognitif erratique. Un exemple de ce profil est le développeur C6E (figures 6.23 et 6.24). On remarque, pour les premiers 35% d'effort, une production d'artefacts entrecoupée de relativement longues acquisitions, suivie d'une production de code source entrecoupée à la fois d'acquisition et de production d'artefacts. Cette signature témoigne d'un détachement du projet.

6.5.3.2 Tendances globales du séquencement cognitif

Outre les 4 profils déterminés, l'analyse du séquencement cognitif permet de découvrir deux tendances globales dignes de mention, soit l'approche opportuniste et le besoin continu de vérification et de validation.

D'une part, en observant n'importe quelle signature de séquencement cognitif, bien que la figure 6.19 du développeur C8E soit particulièrement explicite à ce sujet, on remarque l'existence de l'approche opportuniste, à l'effet que l'acquisition se fait dans une perspective juste à temps. En d'autres mots, le développeur acquiert les connaissances qu'il croit nécessaires, au moment qu'il juge opportun. Cette observation confirme l'irréalisme du modèle de processus en cascades, introduit par Royce. En effet, il n'est pas possible pour un développeur d'acquérir toutes les connaissances nécessaires à un projet dès le début. Ainsi, les modèles de processus incrémental et itératif sont beaucoup plus proches de la réalité de développement logiciel.

D'autre part, en observant n'importe quelle vue A-CR-VV, bien que la figure 6.22 du développeur C7C soit particulièrement explicite à ce sujet, on remarque le besoin continu de vérification et validation au sein d'un projet. En effet, tout développeur doit constamment s'assurer que la cristallisation et la réalisation sont conformes aux règles établies, d'où le très grand nombre d'oscillations entre CR (cristallisation et réalisation) et VV (vérification et validation) sur toute vue A-CR-VV. Il s'agit d'un comportement qui, bien qu'intuitif, n'avait jamais été démontré pour l'ensemble d'un projet.

6.5.4 Relation avec le code source

Un des aspects intéressants de la caractérisation de l'effort concerne l'analyse de sa relation avec le code source réalisé. Afin d'établir cette relation, une mesure de productivité du code source s'avère nécessaire. À cet effet, la mesure du nombre de déclarations (exécutables et déclaratives) est plus précise que la mesure du nombre de lignes de code, car elle est moins influencée par le style de programmation des développeurs. Le tableau 6.14 présente, pour les projets C6, C7 et C8, le nombre de déclarations, l'effort de réalisation, le ratio de déclarations par heure (calculé en divisant le nombre de déclarations par l'effort de réalisation) et l'effort d'acquisition.

Tableau 6.14 : Réalisation de code source et effort d'acquisition

Projet	Déclarations	Effort de réalisation (h)	Ratio de déclarations par heure	Effort d'acquisition (% d'effort total)
C6	3462	228	15	15
C7	4268	143	30	8
C8	3842	180	21	11

Comme on peut le remarquer au tableau 6.14, le ratio de déclarations par heure varie du simple au double (15 à 30), inversement de l'effort d'acquisition (15% à 8%). La figure 6.25 illustre cette relation linéaire qui possède une forte corrélation. Conséquemment, il est raisonnable d'affirmer que l'effort d'acquisition a un impact majeur sur la productivité d'une équipe en terme de réalisation de code source.

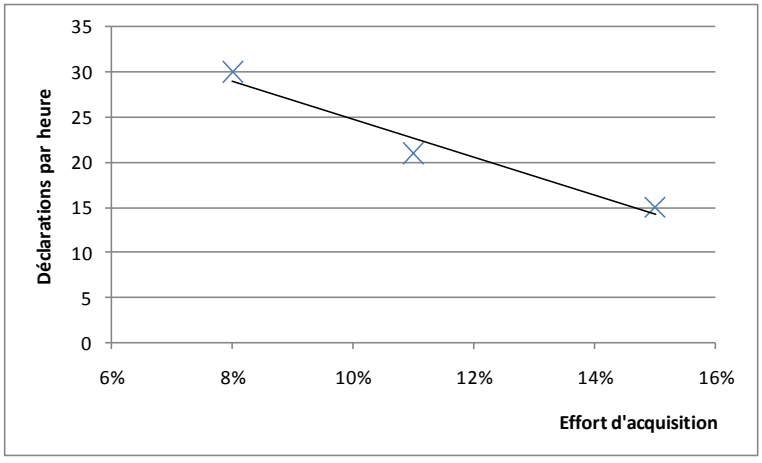

Figure 6.25 : Corrélation entre la réalisation de code source et l'effort d'acquisition

6.6 Discussion

L'objectif de ce chapitre était de caractériser les projets intégrateurs C6, C7 et C8. Il s'agit en fait d'une étude de cas multiples de type exploratoire reposant sur la méthodologie ATS et la modélisation par flux de connaissances (cf. chapitre 3). Ainsi, pour permettre l'analyse des projets, leurs jetons ont été codifiés comme un des 6 facteurs cognitifs du modèle: acquisition, cristallisation, validation, réalisation, vérification ou organisation du travail.

Au cours de la partie exploratoire de la recherche, l'effort s'est avéré être le vecteur le plus prometteur de caractérisation. L'effort a donc été étudié sous plusieurs angles soient la répartition de l'effort global, la répartition et l'évolution du travail individuel et participatif, le séquencement cognitif, ainsi que la relation entre l'effort et le code source.

En ce qui a trait à l'analyse de l'effort global, trois tendances ressortent:

1. La vérification et validation d'un projet comptent pour un peu plus du tiers de l'effort total. En d'autres mots, plus du tiers d'un projet est consacré à assurer la conformité des artefacts et du code source.
2. La cristallisation et la réalisation d'un projet comptent pour un peu moins de la moitié de l'effort total. Autrement dit, près la moitié d'un projet est consacré à élaborer des artefacts et à coder du code source.
3. La cristallisation est environ deux fois plus importante que la validation en terme d'effort. Par contre, la réalisation et la validation ont environ la même importance. Toute proportion gardée, deux fois plus d'effort est consacré à s'assurer de la conformité du code source, par rapport aux artefacts.

Concernant la répartition et l'évolution du travail individuel et participatif, sept tendances ressortent:

1. L'effort individuel d'un projet est d'environ les deux tiers, alors que l'effort participatif est d'environ le tiers de l'effort total. Cette affirmation est vraie seulement si la programmation par paire n'est pas pratiquée par une équipe de développeurs.
2. L'acquisition est très majoritairement individuelle, est investie majoritairement dans la première moitié du projet et évolue de manière opportuniste dans une perspective juste à

temps. De plus, son importance relative est variable selon la différence entre les connaissances déjà acquises par les développeurs d'une équipe avant le début du projet et les connaissances qui seront nécessaires dans le cadre du projet.

3. La cristallisation comporte principalement deux phases. Une phase autant individuelle que participative où les développeurs cristallisent "l'image de possibilité" du logiciel en devenir et une deuxième phase, individuelle, de *retrofitting* des artefacts.
4. La validation est intimement liée à la cristallisation, est majoritairement participative et elle est concentrée dans la première moitié du projet.
5. La réalisation est très majoritairement individuelle (sauf dans le cas de programmation par paire) et est concentrée entre 40% et 90% d'avancement du projet. En fait, le comportement majoritaire des développeurs suivant un processus discipliné consiste à compléter la cristallisation des artefacts avant de commencer la réalisation. Or, la cristallisation n'étant qu'une "image de possibilité" et pas nécessairement la solution qui sera implémentée, cette approche n'est pas optimale, puisqu'elle mènera à un décalage entre la conception et le produit final (cf. chapitre 5).
6. La vérification est intimement liée à la réalisation, est majoritairement individuelle (sauf dans le cas de programmation par paire) et est concentrée dans la deuxième moitié du projet.
7. L'organisation du travail est virtuellement uniquement participative (3 développeurs et plus) et est plus important lors des premiers 30% d'avancement, ce qui est dû au besoin plus important de s'organiser en début de projet. Ce facteur cognitif offre une indication sur la dynamique interne de l'équipe, notamment le degré de maillage d'une équipe et le type de leadership exercé.

Le séquencement cognitif est une contribution significative, à l'effet qu'il n'existe aucune référence dans la littérature relativement à son application sur la totalité d'un projet, plutôt que des périodes de quelques heures. Il en résulte l'observation de quatre profils de développeurs:

1. Le cristallisateur, qui investit la majorité de son temps en production d'artefact, soit en cristallisation ou en validation;

2. Le codeur, qui investit globalement un effort très important en production de code source, soit la réalisation et la vérification;
3. Le polyvalent, qui investit ses efforts selon les besoins du projet;
4. L'agent libre, qui est détaché du projet, ce qui implique un comportement cognitif erratique.

Une telle compréhension des profils naturels des développeurs pourra faciliter la détermination de rôles au sein d'une équipe.

Dans un autre ordre d'idées, en analysant la relation entre la production de code source et l'effort d'un projet, il a été déterminé qu'il existait une corrélation entre la réalisation de code source et l'effort d'acquisition. En effet, plus l'effort d'acquisition est grand, plus la productivité de réalisation de code source est faible. Ce phénomène s'explique par la complexité cognitive de l'acquisition. En effet, l'assimilation efficace de connaissances explicites en connaissances implicites est nécessaire avant la réalisation de code source (qui est en fait transformation de connaissances tacites en connaissances explicites). Un développeur qui aurait déjà assimilé les connaissances nécessaires à la réalisation serait avantagé en termes de productivité, comparativement à un développeur qui doit effectuer l'acquisition de manière opportuniste.

Aucunes données concernant le lien entre les besoins d'acquisition et la réalisation de code source n'existe présentement dans la littérature en génie logiciel. Il s'agit donc d'une contribution importante.

CHAPITRE 7

DISCUSSION GÉNÉRALE

Les travaux présentés dans cette thèse offrent de multiples contributions relativement à la compréhension de l'aspect cognitif du développement logiciel et plus précisément sous une perspective de flux de connaissances. Les contributions sont de trois ordres: méthodologiques, théoriques et pratiques.

7.1 Contributions méthodologiques

Une contribution importante de cette thèse a trait à la méthodologie ATS, présentée en détail dans l'article méthodologique du chapitre 3. Le principal avantage de la méthodologie ATS est qu'elle permet l'analyse du développement logiciel sous une perspective différente de ce qui est possible avec les autres méthodologies utilisées en développement logiciel. En l'occurrence, elle permet la saisie de données cognitives menant à la compréhension du développement logiciel selon une perspective de flux de connaissances.

Comme présenté à la section 3.3.2, les techniques de collection de données peuvent être classées en trois catégories, selon le degré de contact humain requis. La technique utilisée dépend bien sûr de l'objectif de mesure et le choix exige un compromis entre la précision des données recueillies et l'effort d'analyse des données. Ainsi, la méthodologie ATS est une technique de premier degré qui a été développée pour pallier à l'absence de technique de saisie de données cognitives pour la durée totale d'un projet.

Deux techniques de collection de données attirent l'attention, car elles sont utilisées dans le cadre de problématiques complémentaires à cette thèse. Cherry & Robillard (2009) utilisent l'enregistrement audio-vidéo, dans le cadre d'une observation participative, afin de caractériser les interactions ad hoc au sein d'une équipe de développement logiciel. Or, la technique retenue, étant donné sa lourdeur d'analyse, oblige à procéder par échantillonnage, soit dans ce cas 34 heures d'enregistrements audio-vidéo. Par ailleurs, Coman, Sillitti, & Succi (2009) utilisent un système AISEMA (*Automated In-process Software Engineering Measurement and Analysis*) afin

de parvenir à comprendre la nature des interactions entre un développeur et son ordinateur, au cours du développement logiciel. Cette technique de collection de données de troisième degré a l'avantage d'être non intrusive, tout en permettant de recueillir des données sur toute la durée du projet. Or, bien que cette technique permette une analyse comportementale des développeurs, l'aspect cognitif ne peut qu'être déduit, ce qui est caractéristique des techniques de troisième degré. En somme, les techniques de collecte de données utilisées en génie logiciel varient selon l'objectif de mesure et exigent un compromis entre la précision des données recueillies et l'effort d'analyse des données.

En ce qui a trait à la précision et à la validité des jetons d'activité (ATS), la granularité offre une indication sur la rigueur des développeurs et donc sur la fiabilité des jetons. D'abord, l'utilisation de jetons d'activité comme technique de collecte de données est plus fiable et précise que l'utilisation de "feuilles de temps" (*work diary*). En effet, alors que les feuilles de temps s'intéressent principalement à la durée de tâches, les jetons d'activités visent à enregistrer les efforts réels déployés dans des activités cognitives. De plus, afin d'assurer une bonne fiabilité des jetons, les jetons d'activité ont été validés régulièrement, en cours de projet, en s'assurant de leur complétude et de leur représentativité auprès des développeurs. Finalement, une validation de la cohérence des jetons a été effectuée à la fin des projets.

La fiabilité des jetons, compte tenu du facteur humain, qui peut être évaluée entre 80% (profil γ) et 95% (profil α) selon les développeurs, avec une moyenne de 85% à 90% selon l'équipe, a été prise en compte au sein de ces travaux de recherche. En effet, la grande majorité des analyses s'intéresse à des phénomènes plus généraux tels que la caractérisation de profils, de signatures et de tendances, plutôt qu'à des phénomènes de granularité fine.

Finalement, la méthodologie ATS a été utilisée lors de projets intégrateurs en génie logiciel. Dans ce contexte, la méthodologie offre l'avantage supplémentaire de sensibiliser les développeurs à ce qu'ils font. En effet, les développeurs devant consigner dans des jetons d'activités (ATS) tout effort de développement logiciel, ils doivent nécessairement prendre conscience de ce qu'ils font, ce qui constitue un apport méthodologique supplémentaire.

7.2 Contributions théoriques

La contribution théorique la plus importante de cette thèse est certainement le développement d'un modèle de flux de connaissances. À ce propos, les trois articles des chapitres 3, 4 et 5 constituent autant d'exemples d'utilisation pertinente du modèle de flux de connaissances, permettant de mieux comprendre le développement logiciel sur le plan cognitif.

Il est pertinent de rappeler que le modèle de flux de connaissances est une théorie à base empirique (*grounded theory*). Le modèle a été élaboré, au gré de multiples itérations, à partir des données des jetons d'activités. À ce sujet, il est possible de suivre en partie l'évolution du modèle en comparant les deux articles de conférence (cf. annexes A et B) et le modèle "final" présenté dans les articles des chapitres 3, 4 et 5.

Outre les contributions théoriques déjà discutées dans les articles, la forte corrélation observée entre un effort d'acquisition élevé et une productivité du code source faible est particulièrement porteuse. En effet, une telle corrélation identifiée à l'aide d'une étude de cas multiples (3 cas) mérite d'être vérifiée à plus grande échelle. Des modèles prédictifs d'estimation d'effort pourraient éventuellement être développés grâce à cette découverte.

7.3 Contributions pratiques

Étant donné que les expérimentations ont été faites dans le cadre de projets intégrateurs, les contributions pratiques permettront d'abord et avant tout d'améliorer ces projets, et ce, sous quatre aspects: la formation des équipes, le choix du projet, le choix du processus et la supervision des équipes.

7.3.1 Formation des équipes

L'identification de quatre profils de développeur (cristallisateur, codeur, polyvalent, agent libre) au sein des projets étudiés permettra d'orienter la formation des équipes. Notamment, la complémentarité des profils dans une équipe est un aspect à considérer. À cet effet, une bonne compréhension des profils naturels des développeurs pourra faciliter la détermination de responsabilités au sein d'une équipe.

7.3.2 Choix du projet

Les besoins d'acquisition d'une équipe par rapport à un projet constituent un facteur capital à considérer lors du couplage d'une équipe avec un projet intégrateur. En effet, il est crucial de bien évaluer les besoins d'acquisition, soit la différence entre le savoir et le savoir-faire nécessaires au développement d'un projet comparativement au savoir et au savoir-faire que possède préalablement une équipe. Des exemples de projets intégrateurs des dernières années démontrent que des besoins d'acquisition trop importants mènent à un échec du projet ou vers la livraison d'un produit ne répondant pas aux attentes du client.

7.3.3 Choix du processus

À la lumière des résultats présentés dans cette thèse, les développeurs devraient se voir proposer un processus logiciel favorisant le partage de connaissances.

Plus précisément, il est suggéré:

- D'inclure une boucle de validation de l'acquisition, ce qui constitue une généralisation de la pratique de validation de code réutilisation, présentée au chapitre 4. En effet, étant donné l'importance cruciale de l'acquisition dans le développement logiciel, une telle validation serait bénéfique à l'équipe.
- De promouvoir la validation d'artefacts et la vérification de code source de manière participative, possiblement par l'introduction de programmation par paire. Il s'agit d'un moyen privilégié de partage de connaissances. Par ailleurs, l'introduction de séances de revue de code permettrait aux développeurs de détecter des défauts et d'assurer une uniformité dans le code, tout en synchronisant en équipe les connaissances relatives au produit développé.
- Étant donné que les développeurs sous-estiment souvent l'importance de la boucle de rétroaction dans les moments critiques d'un projet, d'insister sur l'importance de la synchronisation et de la validation de connaissances, en particulier lors d'activités antérieures au code, soit principalement lors de la formalisation des exigences, de l'élaboration de l'architecture et de la conception.

- De bien faire comprendre que la conception n'est qu'une image de possibilités, ce qui implique que l'élaboration d'une conception "parfaite" n'a aucun intérêt puisqu'il est très probable que l'implémentation prendra ses distances de cette image de possibilités.

7.3.4 Supervision des équipes

Finalement, bien qu'ils soient généralement peu populaires auprès des développeurs, les jetons d'activité sont très utiles dans une perspective de supervision des équipes. En effet, les jetons fournissent au superviseur des informations détaillées de l'effort investi, facilitant ainsi le suivi de projet. Par ailleurs, comme mentionnée précédemment, l'obligation pour les développeurs de remplir les jetons d'activité les sensibilise à ce qu'ils font.

CONCLUSION ET RECOMMANDATIONS

Les contributions originales de cette thèse à l'avancement des connaissances concernant l'aspect cognitif du développement logiciel sont nombreuses.

D'abord, la méthodologie ATS est détaillée, permettant la saisie de données cognitives et ainsi facilitant l'analyse du développement logiciel selon une perspective de flux de connaissances. En effet, le modèle de flux de connaissances est une théorie à base empirique (*grounded theory*), qui a été élaborée à partir des données des jetons d'activités (ATS), en se basant sur le modèle de création de connaissance de Nonaka & Takeuchi (1995). Un tel modèle facilite la compréhension du développement logiciel sur le plan cognitif.

Puis, le modèle de flux de connaissances est mis à profit afin d'analyser les conséquences de la qualité de documentation dans le cadre de la réutilisation de composants FLOSS. Il en résulte que la documentation incomplète ou inexistante de certains composants FLOSS constitue un danger potentiel à leur réutilisation. Pour pallier à ce risque, une pratique de validation de code réutilisable est proposée.

De plus, les discordances entre les artefacts de conception et l'implémentation de solution sont étudiées d'un point de vue cognitif. Il en résulte que la conception est une discipline opportuniste, majoritairement participative et incluant principalement trois activités cognitives soit l'acquisition de connaissances à partir de sources externes, la cristallisation des connaissances dans des artefacts, ainsi que la validation des connaissances cristallisées, par l'inspection des artefacts. Ainsi, les discordances entre les artefacts de conception et l'implémentation s'expliquent par le fait que la conception n'est qu'une image de possibilités.

Par ailleurs, trois projets intégrateurs sont caractérisés relativement à la production de jetons et à l'effort investi. D'une part, l'analyse des jetons d'activité (ATS) permet de porter un jugement sur la rigueur des développeurs et donc sur la fiabilité des jetons, selon les profils α, β et γ. D'autre part, les facteurs cognitifs sont caractérisés selon leur caractère individuel et participatif. À cet effet, l'acquisition est très majoritairement individuelle et est investie majoritairement dans la première moitié du projet. La cristallisation comporte principalement deux phases. Une phase autant individuelle que participative où les développeurs cristallisent "l'image de possibilité" du

logiciel en devenir et une deuxième phase, individuelle, de *retrofitting* des artefacts. La validation est intimement liée à la cristallisation, est majoritairement participative et elle est concentrée dans la première moitié du projet. La réalisation est très majoritairement individuelle. La vérification est intimement liée à la réalisation, est majoritairement individuelle et est concentrée dans la deuxième moitié du projet. L'organisation du travail est virtuellement uniquement participative. Par ailleurs, le séquencement cognitif permet l'identification de quatre profils de développeurs: le cristallisateur, le codeur, le polyvalent et l'agent libre. Finalement, une forte corrélation a été observée entre un effort d'acquisition élevé et une productivité du code source faible, ce qui constitue une contribution majeure, de par son originalité et ses conséquences théoriques et pratiques.

La principale limitation de cette thèse est au niveau de sa validité externe. En effet, l'expérimentation étant basée sur des projets intégrateurs développés par des étudiants, il est légitime de se questionner sur la validité des résultats dans d'autres conditions, notamment en milieu industriel. À ce sujet, Höst et al. (2000) concluent qu'il n'existe que des différences mineures entre les étudiants de dernière année de baccalauréat et les professionnels, concernant leur habileté à effectuer des tâches relativement simples requérant un jugement. De plus, Porter et al. (1995) ont obtenu des résultats similaires dans une étude des méthodes de détection pour l'inspection des requis logiciels conduits avec des étudiants et répliqués avec des professionnels (Porter & Votta, 1998).

Face à cette limitation potentielle, il est recommandé de conduire les mêmes expérimentations dans un contexte industriel, de manière à prouver la validité externe des résultats.

Une autre avenue de recherche recommandée concerne l'extension de la méthodologie. En effet, il a été démontré que la méthodologie ATS permet d'analyser le développement logiciel dans une perspective de flux de connaissances et il serait très intéressant d'observer la symbiose de cet aspect avec d'autres problématiques complémentaires telles que les interactions ad hoc au sein d'une équipe de développement logiciel et la nature des interactions entre un développeur et son ordinateur, au cours du développement logiciel. Ainsi, les développeurs devraient remplir des jetons d'activité, seraient enregistrés par des caméras audio-vidéo et un système AISEMA recueillerait des données comportementales. Cet amalgame de mesures a le potentiel d'offrir des données permettant de porter à un autre niveau la recherche empirique en génie logiciel.

LISTE DES RÉFÉRENCES

Aaen, I. (2003). Software process improvement: Blueprints versus recipes. *IEEE Software, 20*(5), 86-93.

Alavi, M., & Leidner, D.E. (2001). Review: Knowledge Management and Knowledge Management Systems: Conceptual Foundations and Research Issues. *Management Information Systems Quaterly, 25*(1), 107-136.

Alshayeb, M., & Li, W. (2003). An Empirical Validation of Object-Oriented Metrics in Two Iterative Processes. *IEEE Transactions on Software Engineering, 29*(11), 1043-1049.

Antoniol, G., Caprile, B., Potrich, A., & Tonella, P. (2000). Design Code Traceability for OO Systems. *Annals of Software Engineering, 9*(1-4), 35-58.

Arent, J., & Nørbjerg, J. (2000). Software Process Improvement as Organizational Knowledge Creation: A Multiple Case Analysis. *Proceedings of the 33rd Hawaii International Conference on System Sciences, Maui, Hawaii* (Vol. 4, pp. 4045-4055). Washington, DC, USA: IEEE Computer Society

Argyris, C., & Schon, D. A. (1978). *Organizational Learning: A Theory of Action Perspective*. Reading, MA, USA: Addison-Wesley.

Baetjer, H. J. (1998). *Software as Capital: An Economic Perspective on Software Engineering*. Piscataway, NJ, USA: IEEE.

Bailetti, A. J., & Liu, J. (2003). Comparing software development processes using information theory. *Portland International Conference on Management of Engineering and Technology (PICMET'03),* (pp. 309-315). Portland State University.

Baudrya, B., & Le Traon, Y. (2005). Measuring Design Testability of a UML Class Diagram. *Information and Software Technology*, *47*(13), 859-879.

Beck, K. (1999a). Embracing change with extreme programming. *Computer,* 32(10), 70-77.

Beck, K. (1999b). *Extreme programming explained: Embrace change*. Reading, MA: Addison-Wesley.

Beck, K., Beedle, M., Bennekum, A. V., Cockburn, A., Cunningham, W., Fowler, M. et al. (2001). *Manifesto for Agile Software Development*. Consulté le 7 août 2006, tiré de http://www.agilemanifesto.org

Bettman, J. R., & Park, C. W. (1980). Effects of Prior Knowledge and Experience and Phase of the Choice Process on Consumer Decision Processes: A Protocol Analysis. *Journal of Consumer Research*, *7*(3), 234-248.

Bjornson, F. O., & Dingsoyr, T. (2005). A Study of a Mentoring Program for Knowledge Transfer in a Small Software Consultancy Company. *Lecture Notes in Computer Science* (Vol. 3547, pp. 245-256). Heidelberg: Springer Verlag.

Bjornson, F. O., & Dingsoyr, T. (2008). Knowledge Management in Software Engineering: A Systematic Review of Studied Concepts, Findings and Research Methods Used. *Information and Software Technology*, *(50)*11, 1055-1068.

Boehm, B., Port, D., & Basili, V. (2002). Realizing the benefits of the CMMI with the CeBASE method. *Systems Engineering, 5*(1), 73-88.

Boehm, B., Port, D., Egyed, A., & Abi-Antoun, M. (1999). The MBASE life cycle architecture milestone package: No architecture is an island. *First Working International Conference on Software Architecture (WICSA1)* (pp. 511-528). San Antonio, TX, USA: Kluwer Academic Publishers.

Böhme, M. (2004). *Using libavformat and libavcodec*. Consulté le 7 octobre 2009, tiré de http://www.inb.uni-luebeck.de/~boehme/using_libavcodec.html

Bonke, J. (2005). Paid Work and Unpaid Work: Diary Information Versus Questionnaire Information. *Social Indicators Research*, *70*(3), 349-368.

Briand, L.C., Wu, J., Lounis, H., (2001). Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs. *Empirical Software Engineering*, *6*(1), 11-58.

Carver, J., Jaccheri, L., Morasca, S., & Shull, F. (2003). Issues in Using Students in Empirical Studies in Software Engineering Education. *Proceedings of the Ninth International Software Metrics Symposium (METRICS'03), Sydney, Australia* (pp. 239-249). Washington, DC, USA: IEEE Computer Society.

Cherry, S., & Robillard, P. N. (2009). Audio-video recording of ad hoc software development team interactions. *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering, Vancouver, Canada* (pp. 13-21). Washington, DC, USA: IEEE Computer Society.

Choi, B., & Lee, H. (2002). Knowledge management strategy and its link to knowledge creation process. *Expert Systems with Applications*, *23*(3), 173-187.

Chong, J., & Siino, R. (2006). Interruptions on Software Teams: A Comparison of Paired and Solo Programmers. *Proceedings of the 20th Anniversary Conference on Computer Supported Cooperative Work (CSCW'06), Banff, Alberta* (pp. 29-38). New York, NY, USA: ACM.

Clark, G. (1999). Evaluation of Written Communication: A Replication Study to Determine accuracy. *Corporate Communications: An International Journal*, *4*(3), 112-120.

Clayton, R., Rugaber, S., & Wills, L. (1998). Dowsing: A Tool Framework for Domain-Oriented Browsing of Software Artifacts. *Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE'98), Honolulu, Hawaii* (pp. 204-207). Washington, DC, USA: IEEE Computer Society.

Cockburn, A. (2002). Agile software development joins the "would-be" crowd. *Cutter IT Journal, 15*(1), 6-12.

Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales, *Educational and Psychological Measurement*, *20*(1), 37-46.

Coman, I. D., Sillitti, A., & Succi, G. (2009). A case-study on using an Automated In-process Software Engineering Measurement and Analysis system in an industrial environment. *Proceedings of the 31st International Conference on Software Engineering, Vancouver, Canada* (pp. 89-99). Washington, DC, USA: IEEE Computer Society.

Corbin, J., & Strauss, A. (1990). Grounded Theory Research: Procedures, Canons and Evaluative Criteria. *Qualitative Sociology*, 13(1), 3-21.

Crosby, P. B. (1979). *Quality is Free: The Art of Making Quality Certain*. New York, NY: MacGraw-Hill.

Czerwinski, M., Horvitz, E., & Wilhite, S. (2004). A Diary Study of Task Switching and Interruptions. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Vienna, Austria* (pp. 175-182). New York, NY, USA: ACM.

Dakhli, S. B. D., & Chouikha, M. B. (2009). The Knowledge-Gap Reduction in Software Engineering. *Proceedings of the Third International Conference on Research Challenges in Information Science, Fès, Morocco*.

Dalcher, D. (2003). Dynamic systems development: towards continuity, growth and evolution. *Tenth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, (pp. 62-71). Huntsville, AL, USA: IEEE Computer Society.

Dall'Agnol, M., Janes, A., Succi, G., & Zaninotto, E. (2003). Lean management - a metaphor for extreme programming? *Fourth International Conference on Extreme Programming and Agile Processes in Software Engineering*, (pp. 26-32). Genova, Italy: Springer-Verlag.

Deming, W. E. (1986). *Out of Crisis*. Cambridge, MA: MIT Center for Advanced Engineering.

Desouza, K. C., Awazu, Y., & Wan, Y. (2006). Factors Governing the Consumption of Explicit Knowledge. *Journal of the American Society for Information Science and Technology*, *57*(1), 36-43.

Dick, B., & Dalmau, T. (1990). *Values in Action: Applying the Ideas of Argyris and Schon*. Brisbane, Australia: Interchange.

Earl, M. (2001). Knowledge Management Strategies: Towards a Taxonomy. *Journal of Management Information Systems, 18*(1), 215-233.

Erickson, J., Lyytinen, K., Keng, S. (2005). Agile modeling, agile software development, and extreme programming: the state of research. *Journal of Database Management*, 16(4), 88-100.

Eveleens, J. L., & Verhoef, C. (2010). The Rise and Fall of the Chaos Report Figures. *IEEE Software, 27*(1), 30-36.

Fioravanto, F., & Nesi, P. (2001). Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems. *IEEE Transactions on Software Engineering, 27*(12), 1062-1084.

Flyvbjerg, B. (2006). Five misunderstandings about case-study research. *Qualitative Inquiry, 12*(2), 219-245.

Fowler, M. (2005). The new methodology [agile methodology]. *Software World*, *36*(1), 3-6.

Fujitsu. (2006). *Macroscope*. Fujitsu. Consulté le 24 octobre 2006, tiré de http://www.fujitsu.com/ca/fr/services/consulting/method/macroscope/

Gendreau, O., & Robillard, P. N. (2007). Knowledge Conversion in Software Development. *Proceedings of the Nineteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'2007), Boston, USA* (pp. 392-395). Knowledge Systems Institute Graduate School.

Gendreau, O., & Robillard, P. N. (2009). Exploring Knowledge Flow in Software Project Development. *Proceedings of the 2009 International Conference on Information, Process, and Knowledge Management, Cancun, Mexico* (pp. 99-104). Washington, DC, USA: IEEE Computer Society.

Germain, E., & Robillard, P.N. (2003). What Cognitive Activities are Performed in Student Projects. *Proceedings of the 16th Conference on Software Engineering Education and Training, Madrid, Spain* (pp. 224-231). Washington, DC, USA: IEEE Computer Society.

Germain, E., & Robillard, P. N. (2005). Engineering-based processes and agile methodologies for software development: a comparative case study. *Journal of Systems and Software, 75*(1-2), 17-27.

Gilgun, J. F. (1992). *Definitions, Methodologies, and Methods in Qualitative Family Research, Qualitative Methods in Family Research*. Thousand Oaks: Sage.

Glaser, B. G., & Strauss, A. L. (1967). *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Chicago, IL, USA: Aldine Publishing.

Henninger, S. (1997). Tools Supporting the Creation and Evolution of Software Development Knowledge. *Proceedings of the International Conference on Automated Software Engineering (ASE'97), Incline Village, Nevada* (pp. 46-53). Washington, DC, USA: IEEE Computer Society.

Herriott, R. E., & Firestone, W.A. (1983). Multisite Qualitative Policy Research: Optimizing Descriptions and Generalizability. *Educational Researcher, 12*(2), 14-19.

Hersen, M., & Barlow, D. H. (1976). *Single-case experimental designs: Strategies for Studying behavior*. New York, NY: Pergamon.

Highsmith, J. (1997). Messy, exciting, and anxiety-ridden: adaptive software development. *American Programmer, 10*(4), 23-29.

Höst, M., Regnell, B., & Wohlin, C. (2000). Using Students as Subjects – A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering, 5*(3), 201-214.

ISO. (2003). *Information technology - Process assessment - Part 2: Performing an assessment*. ISO/IEC 15504-2:2003, Genève, Suisse.

Jung, H. W., & Hunter, R. (2001). The relationship between ISO/IEC 15504 process capability levels, ISO 9001 certification and organization size: an empirical study. *Journal of Systems and Software, 59*(1), 43-55.

Juran, J. M. (1988). *Juran on Planning for Quality*. New York, NY: MacMillan.

Kettunen, P., & Laanti, M. (2005). How to steer an embedded software project: tactics for selecting the software process model. *Information and Software Technology, 47*(9), 587-608.

Kim, J., & Carlson, C. R. (2001). Design Units a Layered Approach for Design Driven Software Development. *Information and Software Technology, 43*(9), 539-549.

Ko, A. J., DeLine, R., Venolia, G. (2007). Information Needs in Collocated Software Development Teams. *Proceedings of the 29th International Conference on Software Engineering,* (pp. 344-353). Washington, DC, USA: IEEE Computer Society.

Koch, S. (2005). Evolution of Open Source Software Systems – A Large-Scale Investigation. *Proceedings of the First International Conference on Open Source Systems, Genova, Italy* (pp. 148-153). New York, NY, USA: John Wiley & Sons, Inc.

Kolb, D. (1984). *Experiential Learning: Experience as the Source of Learning and Development*. Englewood Cliffs, NJ, USA: Prentice Hall.

Kolbe, R. H., Burnett, M. S. (1991). Content-Analysis Research: An Examination of Applications with Directives for Improving Research Reliability and Objectivity. *Journal of Consumers Research, 18*(2), 243-250.

Kruchten, P. (2000). *The Rational Unified Process: An Introduction*. Reading, MA: Addison-Wesley.

LaToza, T. D., Venolia, G., & DeLine, R. (2006). Maintaining Mental Models: A Study of Developer Work Habits. *Proceedings of the 28th International Conference on Software Engineering, Shanghai, China* (pp. 492-501). New York, NY, USA: ACM.

Lethbridge, T. C., Sim, S. E., & Singer, J. (2005). Studying Software Engineers: Data Collection Techniques for Software Field Studies. *Empirical Software Engineering, 10*(3), 311-341.

Levesque, M. (2004). Fundamental issues with open source software development. *First Monday, 9*(4-5).

Li, W., Etzkorn, L., Davis, C., & Talburt, J. (2000). An Empirical Study of Object-Oriented System Evolution. *Information and Software Technology*, *42*(6), 373-381.

Lindvall, M., Basili, V., Boehm, B., Costa, P., Dangle, K., Shull, F. et al. (2002). Empirical Findings in Agile Methods. *Proceedings of Extreme Programming and Agile Methods*, (pp. 197-207).

Medvidovic, N., Grunbacher, P., Egyed, A., & Boehm, B.W. (2003). Bridging Models across the Software Lifecycle. *Journal of Systems and Software*, *68*(3), 199-215.

Melnik, G., & Maurer, F. (2004). Direct Verbal Communication as a Catalyst of Agile Knowledge Sharing. *Proceedings of the Agile Development Conference, Salt Lake City, UT, USA* (pp. 21-31). Washington, DC, USA: IEEE Computer Society.

Merilinna, J., & Matinlassi, M. (2006). State of the art and practice of open source component integration. *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, Dubrovnik, Croatia* (pp. 170-177). Washington, DC, USA: IEEE Computer Society.

Miller, J., (2008). Triangulation as a basis for knowledge discovery in software engineering. *Empirical Software Engineering*, *13*(2), 223-228.

Murphy, G. C., Notkin, D., & Sullivan, K. J. (2001). Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering*, *27*(4), 364-380.

Naur, P., Randell, B. (1969). *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, Garmisch, Allemagne: OTAN.

Nawrocki, J. R., Walter, B., & Wojciechowski, A. (2002). Comparison of CMM Level 2 and extreme programming. *7th European Conference on Software Quality (ECSQ 2002),* (pp. 288-297). Helsinki, Finland: Springer-Verlag.

Neill, C. J. (2003). The extreme programming bandwagon: revolution or just revolting?. *IT Professional, 5*(5), 62-64.

Nerur, S., & Balijepally, V. (2007). Theoretical Reflections on Agile Development Methodologies. *Communications of the ACM, 50*(3), 79-83.

Nerur, S., Mahapatra, R., & Mangalaraj, G. (2005). Challenges of migrating to agile methodologies. *Communications of the ACM, 48*(5), 72-78.

Nonaka, I., & Takeuchi, H. (1995). *The Knowledge-Creating Company – How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press.

NSERC. (2005). Tri-*Council Policy Statement: Ethical Conduct for Research Involving Humans*. Natural Sciences and Engineering Research Council of Canada. Consulté le 12 avril 2009, tiré de http://www.pre.ethics.gc.ca/english/policystatement/policystatement.cfm

Palmer, S. R., & Felsing, J. M. (2002). *A Practical Guide to Feature-Driven Development*. Upper Saddle River, NY: Prentice-Hall.

Paulk, M. C., Curtis, B., Chrissis, M. B., & Weber, C. V. (1993). Capability maturity model, version 1.1. *IEEE Software, 10*(4)**,** 18-27.

Paulson, J.W., Succi, G. & Eberlein, A. (2004). An Empirical Study of Open-Source and Closed-Source Software Products, *IEEE Transactions on Software Engineering*, *30*(4), 246-256.

Perreault, W. D., & Leigh, L. (1989). Reliability of Nominal Data Based on Qualitative Judgments. *Journal of Marketing Research*, *26*(2), 135-148.

Perry, D. E., Staudenmayer, N. A., Votta, L. G. 1994. «People, organizations, and process improvement». *IEEE Software*, 11:4**,** 36-45.

Polanyi, M. (1967). *The Tacit Dimension*. Garden City, NY: Doubleday.

Poppendeick, M., & Poppendeick, T. (2003). *Lean Software Development: An Agile Toolkit for Software Development Managers*. Reading, MA: Addison-Wesley.

Porter, A., & Votta, L. (1998). Comparing detection methods for software requirements inspection: A replication using professional subjects. *Empirical Software Engineering*, *3*(4), 355-380.

Porter, A., Votta, L., & Basili, V. R. (1995). Comparing Detection Methods for Software Requirements Inspection: A replicated experiment. *IEEE Transactions on Software Engineering, 21*(6), 563-575.

Ras, E., & Rech, J. (2008). Improving Knowledge Acquisition in Capstone Projects Using Learning Spaces for Experiential Learning. *21st Conference on Software Engineering Education and Training* (pp. 77-84). Washington, DC, USA: IEEE Computer Society.

Rifkin, S. (2001). Why Software Process Innovations Are not Adopted. *IEEE Software, 18*(4), 110-112.

Rising, L., & Janoff, N. S. (2000). The Scrum software development process for small teams. *IEEE Software, 17*(4), 26-32.

Robillard, P. N. (1999). The Role of Knowledge in Software Development. *Communications of the ACM, 42*(1), 87-92.

Robillard, P.N. (2005). Opportunistic Problem Solving in Software Engineering. *IEEE Software*, *22*(6), 60-67.

Robillard, P. N., Kruchten, P., & d'Astous, P. (2003). *Software Engineering Process with the UPEDU*. Boston: Pearson Education.

Robson, C. (2002). *Real World Research: A Resource for Social Scientists and Practitioner-Researchers* (2e éd.). Oxford: Blackwell Publishers.

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131-164.

Schon, D. A. (1983). *The Reflective Practitioner*. New York: Basic Books.

Schwaber, K., Beedle, M. (2002). *Agile Software Development with Scrum*. Prentice Hall.

Seaman, C. B. (1999). Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, *25*(4), 557-572.

Singer, J., Lethbridge, T., Vinson, N., & Anquetil, N. (1997). An examination of software engineering work practices. *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*. Toronto, Ontario, Canada: IBM Press.

Standish Group. (1994). *Chaos Report*. Standish Group. Consulté le 15 novembre 2006, tiré de http://www.standishgroup.com/sample_research/chaos_1994_1.php

Stapleton, J. (1997). *DSDM, Dynamic Systems Development Method: the Method in Practice*. Reading, MA: Addison-Wesley.

Subramanyan, R., & Krisnan, M. S., (2003). Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Transactions on Software Engineering, 29*(4), 297-310.

Taulavuori, A. (2002). *Component documentation in the context of software product lines*. Espoo, Finland: VTT Publications.

Taylor, S. J., & Bogdan, R. (1984). *Introduction to Qualitative Research Methods*. New York: John Wiley & Sons.

Trandsen, E., & Vickery, K. (1998). Learning on demand. *Journal of Knowledge Management, 1*(3), 169-80.

van Gurp, J., & Bosch, J., (2002). Design Erosion: Problems and Causes. *Journal of Systems and Software, 61*(2), 105-119.

Wang, Y., Court, I., Ross, M., Staples, G., King, G., & Dorling, A. (1997). Quantitative evaluation of the SPICE, CMM, ISO 9000 and BOOTSTRAP. *Third IEEE International Software Engineering Standards Symposium and Forum (ISESS 97)*, (pp. 57-68). Walnut Creek, CA, USA: IEEE Computer Society.

Wenger, E. (1998). *Communities of Practice: Learning, Meaning and Identity*. Cambridge University Press.

Wertsch, J. V. (1985). *Cultural, Communication, and Cognition: Vygotskian Perspectives*. Cambridge University Press.

Williams, R. (2006). Narratives of Knowledge and Intelligence... Beyond the Tacit and Explicit. *Journal of Knowledge Management, 10*(4), 81-99.

Xu, S., Rajlich, V., & Marcus, A. (2005). An Empirical Study of Programmer Learning During Incremental Software Development. *Proceedings of the 4th IEEE International Conference on Cognitive Informatics, Irvine, California* (pp. 340-349). Washington, DC, USA: IEEE Computer Society.

Yin, R. K. (2003). *Case Study Research: Design and Methods*. Thousand Oaks, CA: Sage.

Yuming, Z., & Hareton, L. (2006). Empirical Analysis of Object-Oriented, Design Metrics for Predicting, High and Low Severity Faults. *IEEE Transactions on Software Engineering*, *32*(10), 771-789.

Zettel, J., Maurer, F., Munch, J., & Wong, L. (2001). LIPE: a lightweight process for e-business startup companies based on extreme programming. *Third International Conference on Product Focused Software Process Improvement (PROFES 2001)*, (pp. 255-270). Kaiserslautern, Germany: Springer-Verlag.

Annexe A

Knowledge Conversion in Software Development

Proceeding of the Nineteenth International Conference on Software Engineering (SEKE'2007)

Boston, USA, 2007

Knowledge Conversion in Software Development

Olivier Gendreau, Pierre N. Robillard
Computer engineering department
École Polytechnique de Montréal
Montréal, Québec, Canada
{olivier.gendreau, pierre-n.robillard}@polymtl.ca

Abstract

Software processes can be categorized in two types of approach: engineering-based processes, criticized for restraining creativity, and agile methodologies, criticized for being often unpredictable. This paper proposes a conciliatory view of software processes by analysing human cognitive activities. Our approach, based on the SECI knowledge conversion process, defines eight knowledge conversion types. The approach is then tested on a project developed by a team of undergraduate students enrolled in a capstone project during the 2006 winter semester at École Polytechnique de Montréal. The knowledge perspective of the capstone project mainly stresses the importance of creativity and information sharing in collaborative projects.

1. Introduction

Software processes can be categorized in two types of approach. First, there are engineering-based processes such as Rational Unified Process (RUP), Unified Process for EDUcation (UPEDU) or Model-Based Architecting and Software Engineering (MBASE). These processes are mainly criticized for restraining creativity [1]. Second, there are agile methodologies such as Extreme Programming (XP), Scrum, Dynamic Systems Development Method (DSDM), Adaptive Software Development (ASD), Crystal Methodologies, Lean Development (LD), Feature Driven Development (FDD) and Agile Modeling (AM). These methodologies are mainly criticized for being often unpredictable [2]. In an effort to conciliate these two software process views, an hybrid approach as emerged mainly by mixing known advantages of the two approaches, OpenUP [3] being an example.

The elaboration of a software process implies conformance to different standards, conventions and best practices of the software field. Metamodels such as Software Process Engineering Metamodel (SPEM), OPEN Process Framework (OPF), Software Process Improvement Capability dEtermination for Object-Oriented/Component-Based Software Development (OOSPICE) and LiveNet can be used to assure uniform representation between processes. But in order to assess software quality or maturity, reference models such as ISO 9001, Software Capability Maturity Model (SW-CMM), Capability Maturity Model Integration (CMMI) and ISO/IEC 15504 can be used. Also, software process improvement (SPI) models are available such as Personal Software Process (PSP), Team Software Process (TSP) and Initiating, Diagnosing, Establishing, Acting and Learning (IDEAL).

The SPI field can also be categorized in two approaches : blueprints and recipes [4]. We can see similarities between traditional processes and blueprint SPI as between agile methodologies and recipe SPI. In fact, traditional processes and blueprint SPI insist on process and prescription while agile methodologies and recipe SPI insist on people and adaptation.

By reviewing the software process literature, we can conclude that most of the processes stand in the engineered-based process/agile methodologies debate continuum. Since we know that software development is knowledge-intensive [5], in order to manage this duality, the integration of some knowledge engineering concepts should be a valuable avenue. Recurring problems of feedback loops could be corrected by improving and adding software practices in order to achieve an integrated knowledge management. Of course, some software disciplines are more subject to benefit from this type of approach. In particular, software design is a discipline particularly involved with cognitive synchronisation. This knowledge-centered activity is intended to assure that team-mates share the same mental model, the same representation of concepts [6].

This paper proposes a conciliatory view of software processes by analysing human cognitive activities. To achieve this purpose, we suggest analysing knowledge

conversion in software processes. For now, there is not much literature on that matter which leads us to believe in the originality of the approach.

Section 2 presents useful knowledge concepts related to knowledge conversion. Section 3 details the proposed approach to analyze knowledge conversion in software development and section 4 presents the observations from a capstone project.

2. Knowledge concepts

More than two decades ago, Alvin Toffler [7] predicted the imminence of a society based on knowledge as a source of power. Nowadays, we can state that knowledge actually is a strategic tool for enterprises seeking improved profits [8]. Therefore, knowledge management is clearly an important matter.

Information and knowledge are vital forces in today's organizations [9] and particularly software organizations. In fact, information and knowledge are essential during software development lifecycle, predominantly during design. In this regard, Kahkonen and Abrahamsson [10] demonstrated the link between software processes and knowledge creation.

Knowledge is context-specific, meaning it depends on time and space [11]. Information becomes knowledge when it is interpreted by someone, associate to a context and anchored to one's commitments [12]. We can categorize knowledge in two types: explicit and tacit [13]. Explicit knowledge can be expressed in formal and systematic language. It can be processed and stored relatively easily [14], contrarily to tacit knowledge which is highly personal and hard to formalise. It is deeply rooted into one's actions, experience and values [15].

Nonaka, Toyama and Konno [12] developed a dynamic process enabling an organization to create, maintain and exploit knowledge. The Unified Model of Dynamic Knowledge Creation includes three elements: the SECI process, which is a knowledge creation process through tacit and explicit knowledge conversion; *Ba*, which is the knowledge creation context; and knowledge assets including every organization-specific resources essential to value creation.

An organization creates knowledge from the interaction between tacit and explicit knowledge, called knowledge conversion. There are four types of knowledge conversion: socialisation (from tacit knowledge to tacit knowledge); externalisation (from tacit knowledge to explicit knowledge); combination (from explicit knowledge to explicit knowledge); and internalisation (from explicit knowledge to tacit knowledge) [12]. Socialisation relates to the conversion of new tacit knowledge from past experiences. Externalisation is the process of crystallising knowledge by making tacit knowledge explicit. Combination relates to converting explicit knowledge to more complex or systematic explicit knowledge. Internalisation happens when someone embodies explicit knowledge into tacit knowledge.

In an organizational perspective, in order to create knowledge, it is crucial to put strategies in place. Regarding that matter, Choi and Lee [8] found links between knowledge management and SECI knowledge creation process. They conclude that a human strategy is more appropriate for socialisation while a system strategy is more appropriate for combination. As for externalisation and internalisation, a balanced human-system strategy is more appropriate.

Von Krogh, Nonaka and Aben [16] developed four knowledge management strategies depending on knowledge domain and knowledge process. A knowledge domain includes data, information, explicit knowledge and tacit knowledge. A knowledge process can either be knowledge creation or knowledge transfer.

A knowledge gap is a problem without any known solution. When that occurs, key resources are responsible of gathering data and information and to create the necessary knowledge in order to solve the problem. We can easily relate knowledge gap resolution to software design.

A knowledge strategy consists of using knowledge processes (transfer or creation) to knowledge domains (existing or new) in order to achieve strategic goals such as efficiency or innovation. The optimisation strategy is used to transfer knowledge domains already existing in the organization. The expansion strategy is used to create knowledge based on data, information and knowledge already existing. The appropriation strategy is used to build new knowledge domains with external sources. The exploration strategy gives to one or many teams the responsibility to build new knowledge domains from scratch.

To conclude, as said earlier, literature regarding knowledge management application to software processes is sparse which demonstrates the originality of this paper.

3. Proposed approach

We propose to analyse software development by using an approach based on the SECI process developed by Nonaka, Toyama et Konno [12]. Table I specifies the eight different knowledge (conversion) types.

TABLE I. KNOWLEDGE TYPES

Knowledge type	Conversion details	Description
KH	No conversion involved	Know-how
CTT	Tacit to tacit	Information sharing
TE	Tacit to explicit	Knowledge crystallisation
CTE	CTT and tacit to explicit	Collaborative knowledge crystallisation
EE	Explicit to explicit	Combination, review
CEE	CTT and explicit to explicit	Collaborative combination
ET	Explicit to tacit	Learning
CET	CTT and explicit to tacit	Collaborative learning

In table I, we can see that each of the four SECI types of knowledge conversion can either occur in individual or collective contexts with the exception of CTT. In our view, tacit to tacit knowledge conversion can not be done individually in software development. This is because such a thing involves "philosophical thinking" which is not relevant in software development. Another particularity is that KH (know-how) does not involve any knowledge conversion because it is related to procedural activities. CTT is a team activity used to exchange or synchronize information. TE and CTE involve knowledge crystallisation, which means that information is formalised such as when structured information is written in a document. EE and CEE are related to activities not requiring much creativity (tacit knowledge) such as reviewing artefacts or coding from a detailed design. Finally, ET and CET are related to learning activities, such as training.

Table II details the relation between knowledge type and engineering-based and agile software activities.

TABLE II. KNOWLEDGE TYPES AND PROCESSES ACTIVITIES

Knowledge Type	Engineering-based activity	Agile activity
KH	Execute tests, Manage working environment, Integrate system	
CTT	Conduct a meeting, Discuss	
TE and CTE	Design components, Define architecture, Fix major code defect, Plan project's development, Write an artifact	Code, Refactor (major), Plan project's development
EE and CEE	Code, Review, Fix minor code defect, Debug	Refactor (minor), Review, Debug
ET and CET	Attend a training, Learn	

As it can be seen in Table II, knowledge types are similar between engineering-based and agile activities. An important concern is the fuzziness between various knowledge types. The objective is to figure out, for a given activity, the dominant knowledge type. For instance, coding, depending on the process used, can be perceived as a TE or an EE knowledge type. For engineering-based processes, coding mostly involves translating detailed design into code. Therefore, the dominant knowledge type is TE during the design activity and EE during the coding activity. However, in most agile processes, coding is considered a creative activity involving both design and coding. In such a process, the dominant knowledge type is TE. In addition, EE is less important in agile processes than in engineering-based processes. Therefore, for projects needing massive artefacts production, such as in critical systems development, engineering-based processes are more adequate than agile methodologies.

4. Observations from capstone project

The knowledge type approach is tested on a project developed by a team of five undergraduate students enrolled in a capstone project during the 2006 winter semester. It is an optional project-oriented course offered to senior-year students in computer engineering at École Polytechnique de Montréal. The course's particularity is that the project is defined by an industrial partner, this time an international aeronautic company. The project is based on a business needs document supplied by the industrial partner. An engineer from the participating organization meets the students once a week to guide them in developing the software product. The students follow an engineering-based software process derived from the UPEDU.

The methodology used to measure developers effort is a more elaborate version of the effort time slip method popularized by Perry, Staudenmayer and Votta [17] and improved afterwards by Germain and Robillard [18]. Each time a team member executes a task, she/he must log information in a time slip *token* containing the date, start and end time, participants involved in the task, process details and task description. The aeronautic project contained about 1500 *tokens* for a total effort of over one thousand hours.

Table III presents the knowledge type distribution for the project undertaken by the students based on their time slips.

TABLE III. KNOWLEDGE TYPE DISTRIBUTION

Knowledge type	%
KH	7
CTT	14
TE	19
CTE	4
EE	23
CEE	19
ET	12
CET	2

Table III provides some insight into three types of activities that are basic to any software development processes: collaborative activities, creativity and learning.

First, the importance of the collaborative activities spent on this project is obtained by summing up the four knowledge types that involve information exchange: CTT, CTE, CEE, and CET. It is found that although it is an engineering-based project, almost 40% of the team effort is spent on collaborative activities.

Second, creativity is a major endeavour in a software development project. A first level evaluation of the amount of team effort involved in creative activities in this project is to consider all knowledge types that are initiated by tacit knowledge, which are CTT, CTE and TE. These three tacit knowledge types count for 37% of the total team effort. Interestingly, almost half of the creativity effort is done collaboratively.

Finally, in most projects, some learning is needed unless team members are already expert in the field. Learning is characterised by the conversion of explicit knowledge into tacit knowledge. Some of the learning occurred during discussion (CTT) but it is difficult to evaluate its importance. Consequently, for this project, learning activities count for at least 14% (TE and CTE) of the total team effort.

5. Conclusions and future work

Knowledge type approach, based on the SECI process, provides a cognitive perspective to software engineering activities. It defines eight knowledge conversion types (KH, CTT, TE, CTE, EE, CEE, ET, and CET). By recording effort for each process activities, it is possible to evaluate the knowledge type distribution in a project's development.

The knowledge perspective of the capstone project stresses the importance of creativity and information sharing in collaborative projects. More detailed analyses could provide enough insight to enable the design of practices that will be tailored to the creativity needed in projects. Ongoing research are aiming at measuring the difference between disciplined and extreme processes from a knowledge type perspective.

References

[1] A. J. Bailetti and J. Liu, "Comparing software development processes using information theory," presented at Portland International Conference on Management of Engineering and Technology (PICMET'03), Portland, OR, USA, 2003.

[2] M. C. Paulk, "Extreme programming from a CMM perspective," *Software, IEEE*, vol. 18, pp. 19-26, 2001.

[3] Eclipse Foundation, "Eclipse Process Framework Project (EPF)," 2006.

[4] I. Aaen, "Software process improvement: Blueprints versus recipes," *IEEE Software*, vol. 20, pp. 86-93, 2003.

[5] P. N. Robillard, "The Role of Software in Software Development," *Communications of the ACM*, vol. 42, pp. 87-92, 1999.

[6] P. N. Robillard, "Opportunistic Problem Solving in Software Engineering," *Software, IEEE*, vol. 22, pp. 60-67, 2005.

[7] A. Toffler, *Powershift: Knowledge, Wealth and Violence at the Edge of the 21st Century*. New York: Bantam Books, 1990.

[8] B. Choi and H. Lee, "Knowledge management strategy and its link to knowledge creation process," *Expert Systems with Applications*, vol. 23, pp. 173-187, 2002.

[9] E. Trandsen and K. Vickery, "Learning on demand," *Journal of Knowledge Management*, vol. 1, pp. 169-80, 1998.

[10] T. Kahkonen and P. Abrahamsson, "Digging into the fundamentals of extreme programming building the theoretical base for agile methods," presented at 29th Euromicro Conference, Belek-Antalya, Turkey, 2003.

[11] F. A. Hayek, "The Use of Knowledge in Society," *The American Economic Review*, vol. 35, pp. 519-530, 1945.

[12] I. Nonaka, R. Toyama, and N. Konno, "SECI, ba and leadership: a unified model of dynamic knowledge creation," *Long Range Planning*, vol. 33, pp. 5-34, 2000.

[13] M. Polanyi, "The Tacit Dimension," in *Knowledge in Organizations*. Boston: Butterworth-Heinemann, 1997, pp. 135-146.

[14] R. Williams, "Narratives of knowledge and intelligence ... beyond the tacit and explicit," *Journal of Knowledge Management*, vol. 10, pp. 81-99, 2006.

[15] D. A. Schon, *The Reflective Practitioner*. New York: Basic Books, 1983.

[16] G. Von Krogh, I. Nonaka, and M. Aben, "Making the most of your company's knowledge: A strategic framework," *Long Range Planning*, vol. 34, pp. 421-439, 2001.

[17] D. E. Perry, N. A. Staudenmayer, and L. G. Votta, "People, organizations, and process improvement," *IEEE Software*, vol. 11, pp. 36-45, 1994.

[18] E. Germain and P. N. Robillard, "Engineering-based processes and agile methodologies for software development: a comparative case study," *Journal of Systems and Software*, vol. 75, pp. 17-27, 2005.

Annexe B

Exploring Knowledge Flow in Software Project Development

Proceeding of the 2009 International Conference
on Information, Process, and Knowledge Managment

Cancun, Mexique, 2009

Exploring Knowledge Flow in Software Project Development

Olivier Gendreau and Pierre N. Robillard
Department of Computer and Software Engineering
École Polytechnique de Montréal
Montréal, Canada
{olivier.gendreau, pierre-n.robillard}@polymtl.ca

Abstract — **The intent of this paper is to provide a better understanding of the knowledge flow in software project development. The model presented identifies five knowledge sources linked to five basic cognitive factors. The knowledge flow model is applied on a software project developed by a team of undergraduate students enrolled in a capstone project during the 2008 winter semester at École Polytechnique de Montréal. Five cognitive factors – acquisition, crystallization, realization, synchronization, and validation – are present throughout a collaborative project. The relative effort expended in each of the cognitive factors varies significantly during a project's development. This study presents a new knowledge based perspective for evaluating and measuring software engineering processes.**

Keywords – knowledge flow; knowledge creation model; software development; cognitive factors; case study; capstone project.

I. INTRODUCTION

Knowledge can be categorized into explicit and tacit types [1]. Explicit knowledge can be expressed in formal and systematic language. It can be processed and stored relatively easily [2]. Tacit knowledge is deeply rooted into one's actions, experience, and values [3] and, as such, is highly personal and difficult to formalize.

Nonaka and Takeuchi developed a knowledge creation model [4], which is an organizational knowledge creation process through the interaction between tacit and explicit knowledge, called knowledge conversion. There are four types of knowledge conversion: socialization (from tacit knowledge to tacit knowledge), externalization (from tacit knowledge to explicit knowledge), combination (from explicit knowledge to explicit knowledge) and internalization (from explicit knowledge to tacit knowledge). Socialization relates to the conversion of new tacit knowledge from past experiences. Externalization is the process of crystallizing knowledge by making tacit knowledge explicit. Combination relates to converting explicit knowledge to more complex or systematic explicit knowledge. Internalization happens when someone embodies explicit knowledge into tacit knowledge.

Based on the concepts of knowledge internalization and externalization, Sowe et al. [5] propose a knowledge sharing model in free/open source software development. They analyzed knowledge sharing in the developer and user mailing lists of the Debian project. Their model describes knowledge sharing as a flow between knowledge providers and knowledge seekers, using mailing lists as their knowledge base. Borghoff & Preschi [6] identify the flow of knowledge as one of the four basic components of knowledge management. Thus, better understanding knowledge flow will help improve knowledge management in a project.

The knowledge management literature is abundant regarding knowledge conversion, but is sparse regarding knowledge sources and knowledge flow between these sources. Understanding such concepts would improve our comprehension of project development.

The remainder of this paper is organized as follows: section 2 details our knowledge flow model; section 3 applies our model to a capstone project; and section 4 concludes this paper and suggests future directions.

II. KNOWLEDGE FLOW MODEL

This paper is based on the knowledge creation model [4] and identifies five cognitive factors as the base of the cognitive process. Acquisition is the process of constructing knowledge by collecting information from various sources. Synchronization is the process of sharing information mostly from face-to-face interaction and electronic exchanges. Crystallization and realization are processes formalizing implicit knowledge into explicit knowledge, respectively artifacts and source code. Finally, validation is the process of verifying and validating some explicit knowledge (artifacts and source code).

Table I presents the relationships between the cognitive factors and the knowledge creation model.

Based on the knowledge sharing model [5], we derive the knowledge flow model, which is presented as fig. 1.

TABLE I. COGNITIVE FACTORS DERIVED FROM THE KNOWLEDGE CREATION MODEL

Cognitive factor	From	To
Acquisition	Explicit	Tacit
Validation	Explicit	Explicit
Synchronization	Tacit	Tacit
Realization	Tacit	Explicit (code)
Crystallization	Tacit	Explicit (artifact)

The five ovals in fig. 1 represent knowledge sources. External information can be general or very specific to the project under development. Non project-specific information may come from various sources such as the Internet, a paper or a book. An artifact is a physical representation of knowledge, such as software requirements, design or test plan. Teamwork represents team members involved in the creation of artifacts and source code. Individual tacit knowledge is knowledge built from interactions with other knowledge sources.

The arrows in fig. 1 describe the direction of the knowledge flow between knowledge sources. The acquisition cognitive factor is involved when a developer needs to increase his individual tacit knowledge from external information. The crystallization cognitive factor is the translation of a developer's mental representation of a concept (tacit knowledge) into an artifact (explicit knowledge), such as producing a use-case diagram or an architectural artifact, whereas the realization cognitive factor is the production of source code. The validation cognitive factor involves bidirectional knowledge flow between individual tacit knowledge and explicit knowledge such as an artifact or source code. In order to review an artifact or source code, a developer will first acquire knowledge from it and then might correct it. Reviewing the traceability between requirements and tests is an example related to the validation cognitive factor in software development. Finally, the synchronization cognitive factor is related to every activity involving information sharing inside a development team. It can occur in localized synchronous teamwork (face-to-face meeting), delocalized synchronous teamwork (phone conversation) or even asynchronous teamwork (email, chat conversation). Examples of synchronization include discussing the choice of architecture, peer reviewing detailed design artifacts or formalizing requirements with the client.

III. CASE STUDY: CAPSTONE PROJECT

The knowledge flow model was applied on a project developed by a team of five undergraduate students enrolled in a capstone project during the 2008 winter semester. It was an optional project-oriented course offered to senior-year students in software engineering at École Polytechnique de Montréal. The project was based on requirements supplied by an industrial partner. An engineer from the participating organization met the students once a week to coach them throughout the software product development. The students followed an engineering-based software process derived from the Unified Process for EDUcation (UPEDU) [7].

Table II details the 6 disciplines and the 12 related artifacts included in the process used by the development team.

TABLE II. CAPSTONE PROJECT PROCESS DISCIPLINES AND ARTIFACTS

Discipline	Artifact
Requirements	1. Software requirements specification (SRS) 2. Use-case model 3. User-interface prototype
Analysis & design	1. Use-case realization 2. Architecture & design document
Implementation	1. System build
Tests	1. Test plan 2. Test cases 3. Test results
Project management	1. Iteration Plan 2. Work order
Configuration management	1. Configuration management plan

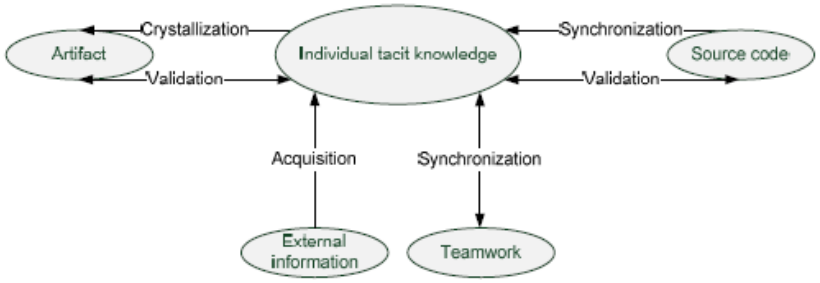

Figure 1. Knowledge flow model

TABLE III. CAPSTONE PROJECT DEVELOPMENT ENVIRONMENT

Component	Tool/technology
Platform	Linux
Programming language	C/C++
Integrated development environment (IDE)	Eclipse
Version control system	GIT
Bug/issue tracking system	TRAC
Communication tools	Skype, xChat, email

The collocated software development team had access to a private development room on campus for the duration of the project, where they had a private workstation, a meeting table and a whiteboard. The main goal of the 2008 winter project was to add videoconferencing capabilities to an existing open-source desktop phone. More specifically, the industrial partner requested 33 functional requirements and 13 non-functional requirements. In order to help the development team to prioritize, functional requirements were classified as essential (24), desirable (8) or optional (1). The requested requirements consisted of 6 new features: two-way video conversation; three-way audio and video conference; audio and video flow synchronization; incoming/outgoing flow encoding/decoding; multiple flow mixing; and H.263, SIP/SDP, RTP, IAX standards compliance.

Table III presents the capstone project development environment. The project was developed in C/C++ on the Linux platform. Many open source tools were used such as GIT (version control system), Eclipse (integrated development environment) and TRAC (project management and bug/issue tracking system). The developers also used Skype, xChat and email as communication tools. The project lasted one semester (13 weeks) and the estimated total team effort was 2000 hours. The developers worked on a fixed schedule that included three half-day sessions per week.

A. *Measurement methodology*

In knowledge management (KM), the process of measurement and development of metrics is made complex by the intangible nature of the knowledge asset [8]. Therefore, the methodology used to measure effort expended by developers in a project is an improved version of Germain & Robillard [9], which is based on the time slip method of Perry, Staudenmayer and Votta [10]. Each time a developer executed a task, information was logged in a time slip token. Each week, an instructor validated tokens for their consistency and reliance. The main fields of a token are: unique identifier; task date; task start time; task end time; task duration (computed from task start/end time); task participants identifier; task main input/output artifacts; process activity identification; and detailed description of the task.

This project produced 1930 validated tokens for a workload of 1813 hours. However, tokens related to academic and technical activities were not accounted for in this analysis, since they were not specific to project development. Academic activities are related to the academic course such as teamwork training, software process concept explanations and project presentation. Technical activities are related to tasks which can be performed by technicians such as configuring the network or setting up and maintaining the development environment. Consequently, there were a total of 1629 development tokens retained for an analysis totaling 1587 hours.

A five category coding scheme was defined to categorize the tokens. For each development token, two independent coders had to decide which of the five cognitive factors was dominant.

B. *Intercoder reliability*

With judgment-based coding schemes, the best approaches for improving the quality of data rely on evaluation of judgments of two (or more) independent coders [11]. The Perreault & Leigh reliability index is preferable to other methods (such as simple percentage of agreement or Cohen's kappa) since it accounts for differences in reliability as a function of the number of categories [12].

$$I_r = [(F_0/N - 1/k)*(k/(k-1))]^{1/2} \quad (1)$$

Equation (1) details Perreault & Leigh's reliability index I_r. F_0 is the observed frequency of agreements between coders, N is the total number of judgments and k is the number of categories.

Table IV shows the results from two independent coders. With a total of 1516 agreements on 1629 judgments and 5 possible categories, the reliability index is 0.96, which indicates a strong reliability of the categorization based on the coding scheme.

TABLE IV. INTERCODER RELIABILITY FOR CAPSTONE PROJECT CODIFICATION

Variable	Value
K	5
N	1629
F_0	1516
I_r	0.96

C. *Results analysis*

The effort analysis from the time slip tokens allows a better understanding of knowledge flow in collaborative software development projects. Fig. 2 presents the cumulative effort expended in each cognitive factor in relation to project completion.

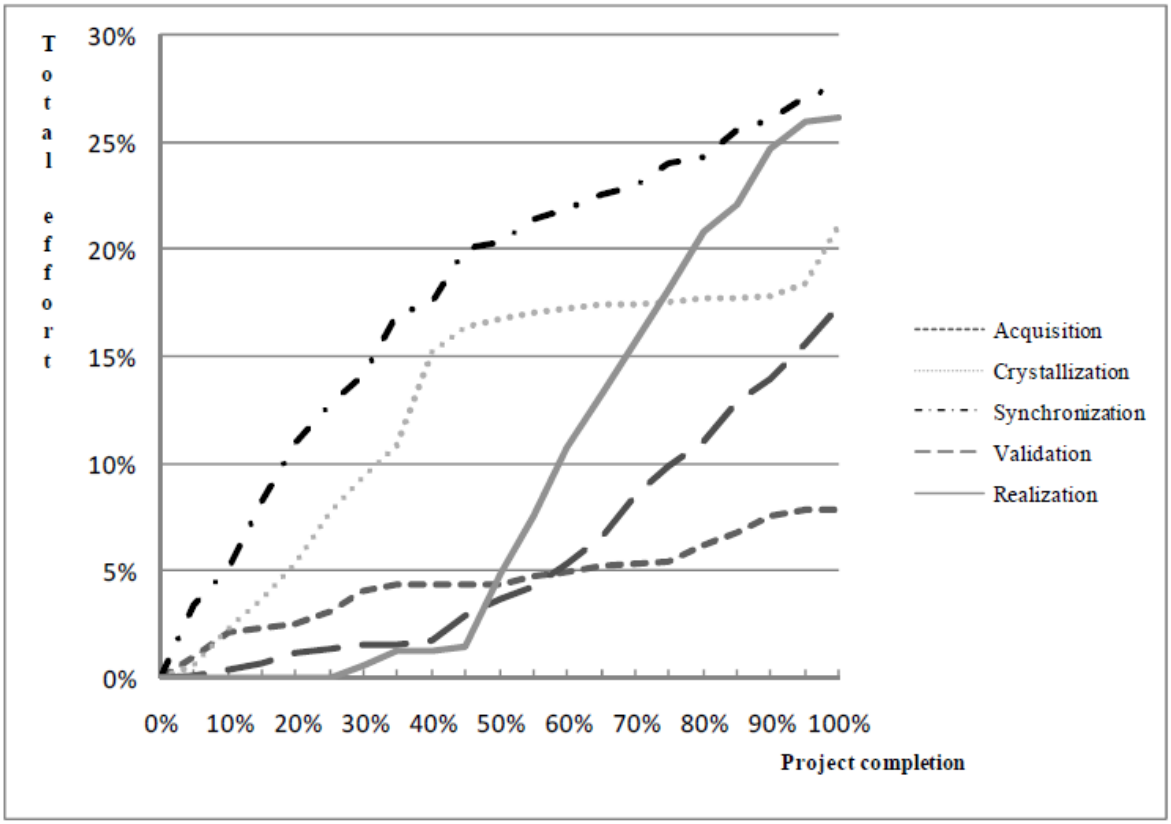

Figure 2. Cumulative cognitive factors effort repartition

Each of the 5 curves of fig. 2 represents the total effort expended (Y-axis) for a given cognitive factor in respect to project completion (X-axis). For example, at 20% of the project completion (X-axis), we observe that the major effort is expended on synchronization with 11% of the total effort (Y-axis), while validation and realization account for respectively 1% and 0% of the total effort. The analysis of the slopes of the 5 curves in fig. 2 allows a better understanding of the relationship between the 5 cognitive factors throughout the project.

Synchronization is the most important cognitive factor, summing up to 28% of the total effort. It appears very important for the first 45% of the project, totaling 20% of the total effort, and then slows down for the rest of the project without stopping to be an important cognitive factor.

Realization is unimportant before 45% of completion, with only 1% of total effort, but requires a major and constant effort from that moment until the end of the project as shown by the almost linear part of the cumulative effort curve (45% to 90% of project completion), ending the project with 26% of total effort.

Crystallization is an important cognitive factor for the first 40% of the project (15% of total effort), then becomes less important until the last 5% of the project, where it jumps from 18% to 21% of effort.

Validation is not significant until 40% of completion, with only 2% of total effort, but requires a significant and constant effort from that moment until the end of the project as shown by the almost linear part of the cumulative effort curve (40% to 100% of project completion), ending the project with 17% of total effort.

Acquisition is somehow important for the first 30% of completion, with 4% of total effort, but slowly progresses until the end the project (8% of total effort).

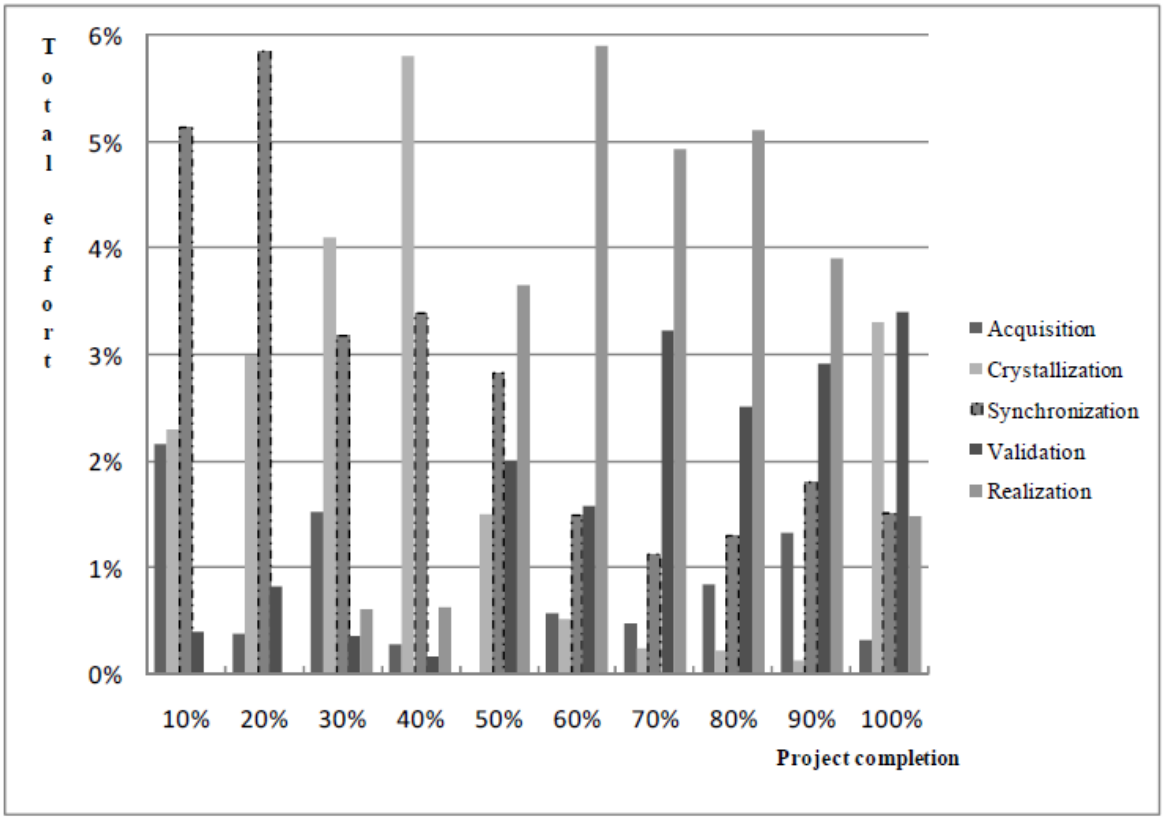

Figure 3. Non-cumulative cognitive factors effort repartition

Fig. 3 illustrates the non-cumulative effort repartition, providing an insight into the relative proportions of cognitive factors throughout the project. This complementary view helps to analyze cognitive factor trends, which are useful for a better understanding of the knowledge flow in software development projects.

D. Discussion

The five cognitive factors detailed in the knowledge flow model (acquisition, crystallization, realization, synchronization and validation) are significant throughout a collaborative project. The relative effort expended in each of the cognitive factors varies according to the project's development life-cycle. For instance, during the requirements phase, the acquisition, synchronization and crystallization effort is much greater than in the construction phase, where the realization and validation effort predominate.

It is noteworthy that crystallization and realization are complementary throughout the project. This is mainly due to the engineering-based process used for the analyzed project development. The objective of this process methodology is the production of artifacts in order to support early decisions regarding requirements and design [9], which explains the observed complementarity. Table V shows that non-cumulative crystallization and realization effort represents about half of the total effort.

TABLE V. NON-CUMULATIVE CRYSTALLIZATION AND REALIZATION EFFORT

Project completion	Crystallization effort	Realization effort	Crystallization and realization effort
10%	2%	0%	2%
20%	3%	0%	3%
30%	4%	1%	5%
40%	6%	1%	6%
50%	1%	4%	5%
60%	0%	6%	6%
70%	0%	5%	5%
80%	0%	5%	5%
90%	0%	4%	4%
100%	3%	1%	5%

TABLE VI. NON-CUMULATIVE CRYSTALLIZATION AND SYNCHRONIZATION EFFORT

Project completion	Crystallization effort	Synchronization effort
10%	2%	5%
20%	3%	6%
30%	4%	3%
40%	6%	3%
50%	1%	3%
60%	0%	1%
70%	0%	1%
80%	0%	1%
90%	0%	2%
100%	3%	2%

Fig. 2, fig. 3, and table VI allow concluding that crystallization and synchronization are closely related. In fact, for the first 40% of project completion, the developers spend much effort on synchronizing requirements, architecture and high-level design. However, from mid-project until the end, both cognitive factors become less important, since the developers now have most of the necessary knowledge required to crystallize artifacts and realize code.

The knowledge flow model confirms the opportunistic aspect of software development [13]. If a developer needs to accomplish a task and is lacking crucial tacit knowledge, he will acquire it from an external source. He might need to synchronize his new tacit knowledge with other developers before crystallizing it as an artifact. Then, another developer can validate the crystallized knowledge. A developer has the ability to easily switch from one cognitive factor to the other.

IV. CONCLUSION

The main contribution of this paper is the identification of five measurable cognitive factors describing as thoroughly as possible knowledge flow in software project development. It allows us to presume that in projects, particularly in software development, activities related to acquisition (related to learning) and synchronization (related to communication) are often underestimated.

One major difficulty of this approach is related to the recording and the codification of the time slip tokens. Rigorous validation of the tokens as the project progresses is needed to maintain some confidence in the data. Intercoder reliability index measurement is required to validate the reliability of the coding scheme.

Future research should intend to further detail the knowledge flow model from various applications. More specifically, the relative importance of various synchronization sources should be analyzed, such as face-to-face meeting, chat, email, and phone conversation.

The results of such studies are likely to help improve the various practices and our understanding of software process activities.

V. REFERENCES

[1] M. Polanyi, "The tacit dimension" in Knowledge in Organizations, L. Prusak (Ed.), Boston: Butterworth-Heinemann, 1997, pp. 135-146.

[2] R. Williams, "Narratives of knowledge and intelligence... beyond the tacit and explicit", Journal of Knowledge Management, vol. 10, 2006, pp. 81-99.

[3] D.A. Schon, The Reflective Practitioner. New York: Basic Books, 1983.

[4] I. Nonaka and H. Takeuchi, The Knowledge-Creating Company - How Japanese Companies Create the Dynamics of Innovation, Oxford University Press, 1995.

[5] S.K. Sowe, I. Stamelos, and L. Angelis, "Understanding knowledge sharing activites in free/open source software projects: An empirical study", Journal of Systems and Software, vol. 81, 2008, pp. 431-436.

[6] U.M. Borghoff and R. Pareschi, Information technology for knowledge management. Berlin: Springer, 1998.

[7] P.N. Robillard, P. Kruchten, and P. d'Astous, Software Engineering Process with the UPEDU, Boston: Addison-Wesley, 2003.

[8] A. Kankanhalli and B.C. Tan, "A review of metrics for knowledge management systems and knowledge management initiatives", Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS'04), Washington, DC, 2004.

[9] E. Germain and P.N. Robillard, "Engineering-based processes and agile methodologies for software development: a comparative case study", Journal of Systems and Software, vol. 75, 2005, pp. 17-27.

[10] D.E. Perry, N.A. Staudenmayer, and L.G. Votta, "People, organizations, and process improvement", IEEE Software, vol. 11, 1994, pp. 36-45.

[11] W.D. Perreault and L. Leigh, "Reliability of nominal data based on qualitative judgments", Journal of Marketing Research, vol. 25, 1989, pp. 135-148.

[12] R.H. Kolbe and M.S. Burnett, "Content-analysis research: an examination of applications with directives for improving research reliability and objectivity", Journal of Consumer Research, vol. 18, 1991, pp. 243-250.

[13] P. N. Robillard, "Opportunistic problem solving in software engineering", IEEE Software, vol. 22, 2005, pp. 60-67.

Annexe C

Échantillon type de jetons ATS

Id	Date	H début	H fin	Effort	P1	P2	P3	P4	P5	It	Art. d'entrée	Art. de sortie	Discipline	Rôle	Activité de processus	Description de l'activité
DM121	2007-02-05	11:40	11:50	00:10		B				3	CPA	CPA	Conception	Ingénieur logiciel	Concevoir les classes	Début de la rédaction du CPA (adaptation du gabarit, écr
DM123	2007-02-05	11:55	12:05	00:10		B				3	CPA	CPA	Conception	Ingénieur logiciel	Concevoir les classes	Rédaction du CPA (suite de l'écriture de la section 1).
DM126	2007-02-05	12:20	12:30	00:10		B				3	CPA	CPA	Conception	Ingénieur logiciel	Concevoir les classes	Rédaction du CPA (suite de l'écriture de la section 1).
DM127	2007-02-05	12:30	12:35	00:05		B			E	3	RCU	Connaissances	Conception	Analyste	Réaliser les cas d'utilisation	Discussion sur la méthode de parsage (conception et imp
PL120	2007-02-05	12:30	12:35	00:05		B			E	3	RCU	Connaissances	Conception	Analyste	Réaliser les cas d'utilisation	Discussion sur la méthode de parsage (conception et imp
DM128	2007-02-05	12:35	12:45	00:10		B				3	CPA	CPA	Conception	Ingénieur logiciel	Concevoir les classes	Rédaction du CPA (suite de l'écriture de la section 1).
LC114	2007-02-05	13:30	14:00	00:30	A					3	ocumentation papi	Connaissances	Conception	Ingénieur logiciel	Formation	Lecture et revision mentale des patron de conception (co
LC115	2007-02-05	14:00	15:00	01:00	A					3	CUI	CPA	Conception	Ingénieur logiciel	Concevoir les classes	Concevoir le diagramme de classe pour le cas critique et
LC116	2007-02-05	15:00	15:30	00:30	A					3	cumentation virtu	Connaissances	Conception	Ingénieur logiciel	Formation	Lecture sur les patron de composite
LC117	2007-02-06	14:00	14:45	00:45	A					3	CPA	Autre	Conception	Ingénieur logiciel	Concevoir les classes	effectuer un diagramme de sequence entre quelques clas
JL83	2007-02-06	14:00	15:00	01:00				D		3	Autre	Connaissances	Formation	Étudiant	Formation	Formation sur le drag and drop
LC118	2007-02-06	14:45	15:45	01:00	A					3	CPA	CPA	Conception	Ingénieur logiciel	Concevoir les classes	Concevoir le diagramme de classe pour le cas critique et
JL84	2007-02-06	15:30	16:30	01:00				D		3	Autre	Prototype	Implémentation	Développeur	Coder les composantes	Travail sur le prototype
JL85	2007-02-06	16:30	17:30	01:00				D		3	Autre	Prototype	Implémentation	Développeur	Coder les composantes	Travail sur le prototype
DM131	2007-02-06	22:00	22:25	00:25		B				3	RCU	cumentation pap	Requis	Analyste	Réaliser les cas d'utilisation	Retranscrire le processus d'analyse des fichiers.
DM133	2007-02-07	10:55	11:10	00:15		B				3	RCU	cumentation pap	Conception	Analyste	Réaliser les cas d'utilisation	Retranscrire le processus d'analyse des fichiers.
DM134	2007-02-07	11:10	12:00	00:50		B				3	Processus	cumentation pap	Gestion	Gestionnaire	Réviser	Dresser la liste des revues à effectuer.
DM135	2007-02-07	12:00	12:20	00:20		B				3	CPA	CPA	Conception	Ingénieur logiciel	Concevoir les classes	Rédaction de la section 1.5 du CPA
DM136	2007-02-07	12:30	13:00	00:30		B				3	CPA	CPA	Conception	Ingénieur logiciel	Concevoir les classes	Rédaction de la section 2.0 du CPA présentation général
DM137	2007-02-07	13:00	13:30	00:30	A	B				3	CPA	CPA	Conception	Ingénieur logiciel	Concevoir les classes	Discussion sur le diagramme de classe du cas d'utilisatio
LC118b	2007-02-07	13:00	13:30	00:30	A	B				3	CPA	CPA	Conception	Ingénieur logiciel	Concevoir les classes	Discussion sur le diagramme de classe du cas d'utilisatio
DM138	2007-02-07	13:30	13:40	00:10		B				3	CUI	CUI	Requis	Analyste	Modéliser les exigences	Correction mineurs sur le CUI (modif. des points d'exten
LC119	2007-02-07	13:50	14:20	00:30	A	B		D	E	3	Discussion	Connaissances	Conception	Ingénieur logiciel	Réviser	Discussion sur la méthode du drag and drop
JL87	2007-02-07	13:50	14:20	00:30	A	B		D	E	3	Discussion	Connaissances	Conception	Ingénieur système	Réviser	Discussion sur la méthode du drag and drop
PL121	2007-02-07	13:50	14:20	00:30	A	B		D	E	3	Discussion	Connaissances	Conception	Ingénieur logiciel	Réviser	Discussion sur la méthode du drag and drop
DM139	2007-02-07	13:50	14:20	00:30	A	B		D	E	3	RCU	Mémo	Conception	Analyste	Réaliser les cas d'utilisation	Discussion sur la méthode du drag and drop
LC120	2007-02-07	14:20	14:40	00:20	A	B		D	E	3	Discussion	Prototype	Conception	Ingénieur logiciel	Réviser	Revue sur la maquette d'interface
JL88	2007-02-07	14:20	14:40	00:20	A	B		D	E	3	Discussion	Prototype	Conception	Ingénieur système	Réviser	Revue sur la maquette d'interface
PL122	2007-02-07	14:20	14:40	00:20	A	B		D	E	3	Discussion	Prototype	Conception	Ingénieur logiciel	Réviser	Revue sur la maquette d'interface
DM140	2007-02-07	14:20	14:40	00:20	A	B		D	E	3	CUI	Mémo	Conception	Analyste	Modéliser les exigences	Discussion préparatoire à la rencontre du client (sujet les
DM141	2007-02-07	14:45	15:00	00:15		B				3	CPA	CPA	Conception	Ingénieur logiciel	Concevoir les classes	Rédaction de la section 2.0 du CPA présentation général
JD120	2007-02-07	14:50	15:00	00:10			C			3	CUI	CUI	Requis	Analyste	Modéliser les exigences	Liste de cas d'utilisation de vue
LC121	2007-02-07	15:05	15:50	00:45	A	B	C	D	E	3	Discussion	Connaissances	Requis	Analyste	Formaliser les besoins	Discussion avec le client sur le parsing et la maquette d'i
JD121	2007-02-07	15:05	15:50	00:45	A	B	C	D	E	3	Discussion	Mémo	Conception	Réviseur	Réviser	Discussion avec le client principalement sur parsage et ir
JL89	2007-02-07	15:05	15:50	00:45	A	B	C	D	E	3	Discussion	Connaissances	Requis	Analyste	Formaliser les besoins	Discussion avec le client sur le parsing et la maquette d'i
PL123	2007-02-07	15:05	15:50	00:45	A	B	C	D	E	3	Discussion	Connaissances	Requis	Analyste	Formaliser les besoins	Discussion avec le client sur le parsing et la maquette d'i
DM142	2007-02-07	15:05	15:50	00:45	A	B	C	D	E	3	Connaissances	Mémo	Requis	Analyste	Réviser	Rencontre avec le client (revue de maquette et du proce
DM143	2007-02-07	15:50	16:00	00:10	A	B				3	CPA	Connaissances	Conception	Ingénieur logiciel	Concevoir les classes	Discussion sur la réalisation du CPA avec Liana
LC122	2007-02-07	15:50	16:00	00:10	A	B				3	Discussion	Connaissances	Formation	Étudiant	Formation	Discussion sur la réalisation du CPA
JD124	2007-02-07	16:20	16:40	00:20			C			3	SRS	SRS	Requis	Ingénieur système	Formaliser les besoins	Modification du SRS suite à la rencontre du client
JD125	2007-02-07	16:40	17:00	00:20	A		C	D	E	3	cumentation virtu	Connaissances	Gestion	Testeur	Planifier les tests logiciels	Discussion quant au test driven, choix de NUnit
PL126	2007-02-07	16:40	17:00	00:20	A		C	D	E	3	PTL	Mémo	Tests	Testeur	Planifier les tests logiciels	Prise de décision sur le choix de l'utilitaire de test
LC124	2007-02-07	16:40	17:00	00:20	A		C	D	E	3	PTL	Mémo	Tests	Testeur	Planifier les tests logiciels	Prise de décision sur le choix de l'utilitaire de test
JL92	2007-02-07	16:40	17:00	00:20	A		C	D	E	3	Discussion	Autre	Tests	Ingénieur système	Planifier les tests logiciels	Prise de décision sur l'utilitaire de test
JD126	2007-02-07	17:00	17:50	00:50			C			3	CUI	CUI	Requis	Analyste	Modéliser les exigences	Début de rédaction des CU de vue, diagramme de CU da
PL127	2007-02-07	17:00	17:20	00:20					E	3	Prototype	Prototype	Implémentation	Programmeur	Corriger les composantes	corriger lancement exécutable
LC125	2007-02-07	17:15	18:15	01:00	A					3	CPA	CPA	Conception	Ingénieur logiciel	Concevoir les classes	Concevoir le diagramme de classe pour le cas critique et
JL93	2007-02-07	17:20	17:50	00:30				D	E	3	Prototype	Autre	Implémentation	Ingénieur logiciel	Coder les composantes	implanter lancement exécutable avec add-in
PL128	2007-02-07	17:20	17:50	00:30				D	E	3	Prototype	Prototype	Implémentation	Programmeur	Coder les composantes	implanter lancement exécutable avec add-in
JD127	2007-02-07	17:50	18:10	00:20			C			3	cumentation virtu	Connaissances	Conception	Réviseur	Réviser	Pris connaissance / Révision du Diagramme de classes d
LC126	2007-02-08	21:00	22:00	01:00	A					3	ocumentation papi	Autre	Gestion	Gestionnaire	érer la configuration du produ	lecture du document du client, le guide de programation.
JD129	2007-02-09	08:35	09:15	00:40			C			3	CUI	CUI	Requis	Analyste	Modéliser les exigences	Continué à détailler les CU de vue
JL95	2007-02-09	09:00	09:30	00:30				D		3	Connaissances	Prototype	Implémentation	Ingénieur logiciel	Coder les composantes	Travail sur le lancement d'une application MS-Dev
JD130	2007-02-09	09:15	09:30	00:15				D	E	3	Discussion	Mémo	Requis	Analyste	Modéliser les exigences	Discussion sur l'interface, comment charger/sauvegarder
PL130	2007-02-09	09:15	09:30	00:15				D	E	3	CUI	CUI	Requis	Analyste	Modéliser les exigences	discussion sur chargement et interface
JD131	2007-02-09	09:30	09:55	00:25			C			3	CUI	CUI	Requis	Analyste	Modéliser les exigences	Continué à détailler les CU de vue
LC129	2007-02-09	09:30	10:00	00:30	A			D	E	3	CPA	Mémo	Conception	Ingénieur logiciel	Réviser	revision et discussion sur le diagramme de classe (memo
JL96	2007-02-09	09:30	10:00	00:30	A			D	E	3	cumentation virtu	Connaissances	Conception	Ingénieur système	Réviser	revision et discussion sur le diagramme de classe
PL131	2007-02-09	09:30	10:00	00:30	A			D	E	3	CPA	Mémo	Conception	Ingénieur logiciel	Réviser	retour sur classes cas critique
LC130	2007-02-09	10:00	10:45	00:45	A					3	CPA	CPA	Conception	Ingénieur logiciel	Autre	redaction de l'artefact CPA
JD133	2007-02-09	10:15	10:45	00:30			C			3	CUI	CUI	Requis	Analyste	Modéliser les exigences	Continué à détailler les CU de vue
PL132	2007-02-09	10:30	11:30	01:00					E	3	CPA	Produit logiciel	Implémentation	Programmeur	Coder les composantes	coder les classes d'analyse de fichiers
JD135	2007-02-09	11:00	11:50	00:50			C			3	CUI	CUI	Requis	Analyste	Modéliser les exigences	Continué (et terminé) le détail les CU de vue
LC133	2007-02-09	11:15	12:15	01:00	A					3	CPA	CPA	Conception	Ingénieur logiciel	Autre	Redaction de l'artefact CPA et mise en page de la sectior